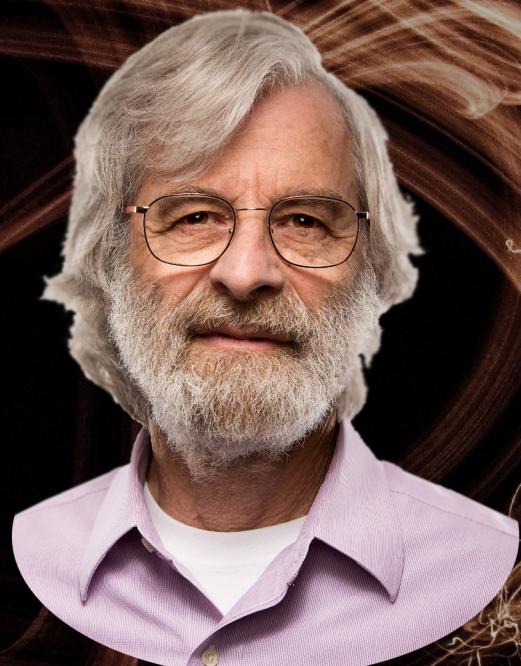


Concurrency

*The Works of
Leslie Lamport*

Dahlia Malkhi
Editor



ASSOCIATION FOR COMPUTING MACHINERY



Concurrency

ACM Books

Editor in Chief

M. Tamer Özsu, *University of Waterloo*

ACM Books is a series of high-quality books for the computer science community, published by ACM and many in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Concurrency: The Works of Leslie Lamport

Dahlia Malkhi, *VMware Research* and *Calibra*
2019

Providing Sound Foundations for Cryptography: On the work of Shafi Goldwasser and Silvio Micali

Oded Goldreich, *Weizmann Institute of Science*
2019

The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!

Ivar Jacobson, *Ivar Jacobson International*
Harold “Bud” Lawson, *Lawson Konsult AB (deceased)*
Pan-Wei Ng, *DBS Singapore*
Paul E. McMahon, *PEM Systems*
Michael Goedicke, *Universität Duisburg-Essen*
2019

Data Cleaning

Ihab F. Ilyas, *University of Waterloo*
Xu Chu, *Georgia Institute of Technology*
2019

Conversational UX Design: A Practitioner’s Guide to the Natural Conversation Framework

Robert J. Moore, *IBM Research-Almaden*
Raphael Arar, *IBM Research-Almaden*
2019

Heterogeneous Computing: Hardware and Software Perspectives

Mohamed Zahran, *New York University*
2019

Hardness of Approximation Between P and NP

Aviad Rubinstein, *Stanford University*

2019

The Handbook of Multimodal-Multisensor Interfaces, Volume 3:

Language Processing, Software, Commercialization, and Emerging Directions

Editors: Sharon Oviatt, *Monash University*

Björn Schuller, *Imperial College London and University of Augsburg*

Philip R. Cohen, *Monash University*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2019

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker

Editor: Michael L. Brodie, *Massachusetts Institute of Technology*

2018

The Handbook of Multimodal-Multisensor Interfaces, Volume 2:

Signal Processing, Architectures, and Detection of Emotion and Cognition

Editors: Sharon Oviatt, *Monash University*

Björn Schuller, *University of Augsburg and Imperial College London*

Philip R. Cohen, *Monash University*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2018

Declarative Logic Programming: Theory, Systems, and Applications

Editors: Michael Kifer, *Stony Brook University*

Yanhong Annie Liu, *Stony Brook University*

2018

The Sparse Fourier Transform: Theory and Practice

Haitham Hassanieh, *University of Illinois at Urbana-Champaign*

2018

The Continuing Arms Race: Code-Reuse Attacks and Defenses

Editors: Per Larsen, *Immunant, Inc.*

Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*

2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun, *University of California, Berkeley*

2017

Computational Prediction of Protein Complexes from Protein Interaction Networks

Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*

Chern Han Yong, *Duke-National University of Singapore Medical School*

Limsoon Wong, *National University of Singapore*

2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations

Editors: Sharon Oviatt, *Incaa Designs*

Björn Schuller, *University of Passau and Imperial College London*

Philip R. Cohen, *Voicebox Technologies*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2017

Communities of Computing: Computer Science and Society in the ACM

Thomas J. Misa, Editor, *University of Minnesota*

2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai, *University of Illinois at Urbana-Champaign*

Sean Massung, *University of Illinois at Urbana-Champaign*

2016

An Architecture for Fast and General Data Processing on Large Clusters

Matei Zaharia, *Stanford University*

2016

Reactive Internet Programming: State Chart XML in Action

Franck Barbier, *University of Pau, France*

2016

Verified Functional Programming in Agda

Aaron Stump, *The University of Iowa*

2016

The VR Book: Human-Centered Design for Virtual Reality

Jason Jerald, *NextGen Interactions*

2016

Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age

Robin Hammerman, *Stevens Institute of Technology*

Andrew L. Russell, *Stevens Institute of Technology*

2016

Edmund Berkeley and the Social Responsibility of Computer Professionals

Bernadette Longo, *New Jersey Institute of Technology*

2015

Candidate Multilinear Maps

Sanjam Garg, *University of California, Berkeley*

2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing

John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*

2015

A Framework for Scientific Discovery through Video Games

Seth Cooper, *University of Washington*

2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers

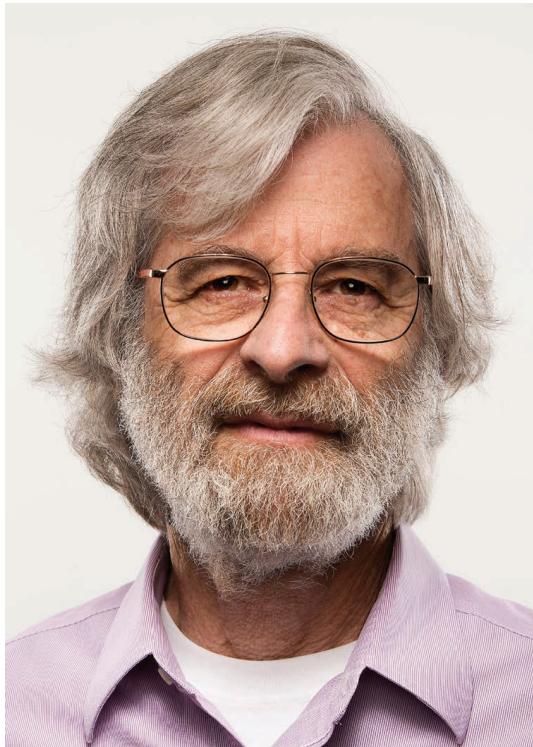
Bryan Jeffrey Parno, *Microsoft Research*

2014

Embracing Interference in Wireless Systems

Shyamnath Gollakota, *University of Washington*

2014



Concurrency

The Works of Leslie Lamport

Dahlia Malkhi, editor

VMware Research and Calibra

ACM Books #29



Copyright © 2019 by Association for Computing Machinery

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews—without the prior permission of the publisher.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which the Association for Computing Machinery is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Concurrency: The Works of Leslie Lamport

Dahlia Malkhi, editor

books.acm.org

<http://books.acm.org>

ISBN: 978-1-4503-7270-1 hardcover

ISBN: 978-1-4503-7271-8 paperback

ISBN: 978-1-4503-7273-2 eBook

ISBN: 978-1-4503-7272-5 EPUB

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3335772 Book	10.1145/3335772.3335781 Chapter 6
10.1145/3335772.3335773 Preface	10.1145/3335772.3335782 Paper 1
10.1145/3335772.3335774 Introduction	10.1145/3335772.3335934 Paper 2
10.1145/3335772.3335775 Paper 0	10.1145/3335772.3335935 Paper 3
10.1145/3335772.3335776 Chapter 1	10.1145/3335772.3335936 Paper 4
10.1145/3335772.3335777 Chapter 2	10.1145/3335772.3335937 Paper 5
10.1145/3335772.3335778 Chapter 3	10.1145/3335772.3335938 Paper 6
10.1145/3335772.3335779 Chapter 4	10.1145/3335772.3335939 Paper 7
10.1145/3335772.3335780 Chapter 5	10.1145/3335772.3335940 References/Index/Bios

A publication in the ACM Books series, #29

Editor in Chief: M. Tamer Özsü, *University of Waterloo*

This book was typeset in Arnhem Pro 10/14 and Flama using ZzTeX.

First Edition

10 9 8 7 6 5 4 3 2 1

Contents

Preface **xvii**

Photo and Text Credits **xix**

Introduction **1**

Dahlia Malkhi, Idit Keidar

Chapter 1: Shared Memory and the Bakery Algorithm **3**

Chapter 2: The Notions of Time and Global State in a Distributed System **5**

Chapter 3: Byzantine Faults **7**

Chapter 4: State Machine Replication with Benign Failures **9**

Chapter 5: Formal Specification and Verification **10**

Chapter 6: Biography **11**

Closing Remarks **12**

The Computer Science of Concurrency: The Early Years **13**

Leslie Lamport

1 Foreword **13**

2 The Beginning: Mutual Exclusion **14**

3 Producer-Consumer Synchronization **17**

4 Distributed Algorithms **23**

5 Afterwards **24**

References **25**

PART I TECHNICAL PERSPECTIVES ON LAMPORT'S WORK **27**

Chapter 1 Shared Memory and the Bakery Algorithm 29

Hagit Attiya, Jennifer L. Welch

1.1 Introduction **29**

- 1.2 Flavors of the Bakery Algorithm **30**
- 1.3 A Plethora of Registers **36**
- 1.4 A New Model for Describing Concurrency **41**
- 1.5 Sequential Consistency **44**

Chapter 2 The Notions of Time and Global State in a Distributed System **47**

Karolos Antoniadis, Rachid Guerraoui

- 2.1 Introduction **47**
- 2.2 The Notion of Logical Time **49**
- 2.3 The Distributed State Machine Abstraction **53**
- 2.4 The Notion of Distributed Global State **58**
- 2.5 Conclusion **65**

Chapter 3 Byzantine Faults **67**

Christian Cachin

- 3.1 Introduction **67**
- 3.2 Byzantine Agreement **68**
- 3.3 Byzantine Clock Synchronization **76**
- 3.4 Digital Signatures **77**

Chapter 4 State Machine Replication with Benign Failures **83**

Robbert van Renesse

- 4.1 Active versus Passive Replication **85**
- 4.2 A Brief Review of State Machine Replication **85**
- 4.3 Benign System Models **87**
- 4.4 SMR Protocol Basics **88**
- 4.5 Early Asynchronous Consensus Protocols **90**
- 4.6 Paxos **96**
- 4.7 Dynamic Reconfiguration **100**

Chapter 5 Formal Specification and Verification **103**

Stephan Merz

- 5.1 Introduction **103**
- 5.2 The Temporal Logic of Actions **105**
- 5.3 The Specification Language TLA⁺ **115**
- 5.4 PLUSCAL: An Algorithm Language **119**

- 5.5 Tool Support **123**
- 5.6 Impact **127**

Chapter 6 Biography **131**

Roy Levin

- 6.1 Early Years **131**
- 6.2 Education and Early Employment **133**
- 6.3 The COMPASS Years (1970–1977) **134**
- 6.4 The SRI Years (1977–1985) **139**
- 6.5 The DEC/Compaq Years (1985–2001) **146**
- 6.6 The Microsoft Years (2001–) **155**
- 6.7 Honors **163**
- 6.8 Collegial Influences **165**

PART II SELECTED PAPERS **171**

A New Solution of Dijkstra's Concurrent Programming Problem **173**

Leslie Lamport

- Introduction **173**
- The Algorithm **174**
- Proof of Correctness **175**
- Further Remarks **177**
- Conclusion **177**
- References **178**

Time, Clocks, and the Ordering of Events in a Distributed System **179**

Leslie Lamport

- Introduction **180**
- The Partial Ordering **180**
- Logical Clocks **182**
- Ordering the Events Totally **185**
- Anomalous Behavior **189**
- Physical Clocks **190**
- Conclusion **192**
- Appendix **193**
- References **196**

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs **197**

Leslie Lamport

References **201**

The Byzantine Generals Problem **203**

Leslie Lamport, Robert Shostak, Marshall Pease

- 1** Introduction **204**
- 2** Impossibility Results **206**
- 3** A Solution with Oral Messages **210**
- 4** A Solution with Signed Messages **213**
- 5** Missing Communication Paths **217**
- 6** Reliable Systems **221**
- 7** Conclusion **225**
- References **226**

The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication **227**

Leslie Lamport

- Abstract **227**
- 1** Introduction **228**
- 2** The Model **228**
- 3** Interprocess Communication **234**
- 4** Processes **239**
- 5** Multiple-Reader Variables **242**
- 6** Discussion of the Assumptions **243**
- 7** Conclusion **244**
- Acknowledgments **244**
- References **244**

The Mutual Exclusion Problem: Part II—Statement and Solutions **247**

Leslie Lamport

- Abstract **247**
- 1** Introduction **248**
- 2** The Problem **249**

3 The Solutions **258**

4 Conclusion **273**

References **275**

The Part-Time Parliament **277**

Leslie Lamport

1 The Problem **278**

2 The Single-Decree Synod **281**

3 The Multidecree Parliament **294**

4 Relevance to Computer Science **304**

Acknowledgments **316**

References **316**

References **319**

Index **335**

Biographies **343**

Preface

This book is a collective work of many contributors. Leslie Lamport gave the world his work. Chapter authors, listed at the beginning of each chapter, volunteered their time and expert knowledge. Additional people contributed comments or portions of chapters, including: Mani Chandy, Vassos Hadzilacos, Jon Howell, Igor Konnov, Daphna Keidar, Butler Lampson, Kartik Nayak, Tom Rodeheffer, Fred Schneider, Yuan Yu, Lidong Zhou. Ted Yin edited the bibliography and helped typeset the biography chapter. Ruth E. Thaler-Carter provided preliminary biographical notes. Mimi Bussam and Fred Schneider provided several photos of Lamport for this book. The ACM and Turing series editor Tamer Özsü initiated this work and provided support and resources for it. Leslie Lamport developed \LaTeX , the typesetting macro environment used for writing this book.



Leslie Lamport at his uncle's wedding, seated in front.
He is about six years old in this photo.



Leslie Lamport at the National Academy of Sciences induction, Washington, April 28, 2011.

Photo and Text Credits

Photos

Page viii Richard Morgenstein Photography, © Association for Computing Machinery, Inc. 2014

Page xviii Photo courtesy of Leslie Lamport

Page xviii Photo courtesy of Leslie Lamport

Page 27 © Dag Johansen, Photo courtesy of Leslie Lamport

Page 28 Photo courtesy of Leslie Lamport

Page 170 Photo courtesy of Leslie Lamport

Page 170 © Keith Marzullo 2001

Page 171 Photo courtesy of Leslie Lamport

Page 172 Photo courtesy of Leslie Lamport

Page 172 Photo courtesy of Leslie Lamport

Page 343 Photo courtesy of Dahlia Malkhi

Text

Page 1 “Introduction” by Dahlia Malkhi and Idit Keider. Copyright © Dahlia Malkhi and Idit Keider. Reprinted by permission of Dahlia Malkhi and Idit Keider.

Page 13 Leslie Lamport. 2015. Turing lecture: The computer science of concurrency: the early years. Commun. ACM 58, 6 (May 2015), 71–76. DOI: <https://doi.org/10.1145/2771951>

Page 29 “Shared Memory and the Bakery Algorithm,” Hagit Attiya and Jennifer L. Welch, Copyright © Hagit Attiya and Jennifer L. Welch. Reprinted by permission of Hagit Attiya and Jennifer L. Welch.

Page 47 “The Notions of Time and Global Sstate in a Distributed System” by Karoloa Antoniadis and Rachid Guerraoui. Copyright © Karoloa Antoniadis and Rachid Geurraoui. Reprinted by permission of Karoloa Antoniadis and Rachid Guerraoui.

Page 67 “Byzantine Faults” by Christian Cachin. Copyright © Christian Cachin. Reprinted by permission of Christian Cachin.

xx Photo and Text Credits

Page 83 “State Machine Replication with Benign Failures” by Robbert van Renesse. Copyright © Robbert van Renesse. Reprinted by permission of Robbert van Renesse.

Page 103 “Formal Specification and Verification” by Stephan Merz. Copyright © Stephan Merz. Reprinted by permission of Stephan Merz.

Page 131 “Biography of Leslie Lamport” by Roy Levin. Copyright © Roy Levin. Reprinted with permission of Roy Levin.

Page 173 Leslie Lamport. 1974. A new solution of Dijkstra’s concurrent programming problem. *Commun. ACM* 17, 8 (August 1974), 453–455. DOI: <https://doi.org/10.1145/361082.361093>

Page 179 Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. DOI: <http://dx.doi.org/10.1145/359545.359563>

Page 197 Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” in *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sept. 1979. DOI: <10.1109/TC.1979.1675439>

Page 203 Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. DOI: <http://dx.doi.org/10.1145/357172.357176>

Page 227 Leslie Lamport. 1986. The mutual exclusion problem: part II—statement and solutions. *J. ACM* 33, 2 (April 1986), 327–348. DOI: <http://dx.doi.org/10.1145/5383.5385>

Page 247 Leslie Lamport. 1986. The mutual exclusion problem: part II—statement and solutions. *J. ACM* 33, 2 (April 1986), 327–348. DOI: <http://dx.doi.org/10.1145/5383.5385>

Page 277 Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. DOI: <https://doi.org/10.1145/279227.279229>

Introduction

Dahlia Malkhi, Idit Keidar

Back in the days when the world's first multiprocessor computers were being built and clouds existed only in the sky, Leslie Lamport ruminated about a bakery. He observed that in order to be served one at a time, customers had to solve a mutual exclusion problem, and discovered a way for them to do so without the baker's help. The resulting Bakery algorithm foreshadowed some of the most important developments in multiprocessor programming for years to come. Lamport's works have since been interwoven with four and a half decades of evolution of digital computing technology, while multiprocessing and distributed computing have become increasingly commonplace.

The body of Lamport's works lays formal foundations for concurrent computations executed by multiple processes—be they threads running on a shared memory multicore platform or autonomous agents communicating via message passing. He put forward fundamental concepts, such as causality and logical time, atomic shared registers, sequential consistency, state machine replication, Byzantine agreement, and wait-freedom. Some of his algorithms have become standard engineering practice for fault-tolerant distributed computing—distributed systems that continue to function correctly despite failures of individual components. He developed a substantial body of work on the formal specification and verification of concurrent systems, and has contributed to the development of automated tools applying these methods.

This book begins with an article covering Lamport's A.M. Turing Award lecture, which he gave at the PODC Conference in 2014. The article was published in the *Communications of the ACM* in 2015. It recalls the cradle of concurrency from 1965 to 1977, with forty years of hindsight. The first concurrent computing problem

2 Introduction

formulated was mutual exclusion. Next came producer-consumer algorithms and their extensions, and then distributed algorithms. With these problems as examples, the article illustrates different approaches to describing concurrent algorithms and to reasoning about them. Lamport dedicated this historical review to Edsger Dijkstra, who played a fundamental role in the study of concurrency.

This introduction is followed by two parts. Part I includes technical chapters centered around five key topics addressed in Lamport's works: Chapter 1, shared memory and the Bakery algorithm for mutual exclusion; Chapter 2, state machines and event ordering; Chapter 3, Byzantine protocols; Chapter 4, fault-tolerant replication; and Chapter 5, formal methods. Each of the chapters presents an expert's retrospective on Lamport's original ideas. They explain how Lamport tackled certain problems and also underscore a number of common themes that recurred throughout his career.

One prominent characteristic of Lamport's work is that to approach specific problems, he develops novel and more general foundations. Time and time again, the experience of devising concurrent algorithms and the challenge of verifying their correctness led him to focus on the basic premises that would enable a mathematical study of multiprocessor behavior. For example, working on the Bakery algorithm (Chapter 1) led Lamport to define the semantics of memory "store" and "load" operations in multiprocessors. Lamport later said, "For a couple of years after my discovery of the Bakery algorithm, everything I learned about concurrency came from studying it" [Lamport and Levin 2016a]. Another example is the Byzantine generals problem, an agreement problem in a model that characterizes faulty (or "buggy") processor behavior. This concept emerged while working on a fault-tolerant multicomputer system for executing avionics software. All in all, many of the abstractions and principles that Lamport invented in order to tackle specific problems ended up becoming theoretical pillars of concurrent programming.

A second theme in Lamport's work is addressing practical problems. Indeed, spending his research career in industrial research environments was not an accident. "I like working in an industrial research lab, because of the input," Lamport said. "If I just work by myself and come up with problems, I'd come up with some small number of things, but if I go out into the world, where people are working on real computer systems, there are a million problems out there. When I look back on most of the things I worked on—Byzantine generals, Paxos—they came from real-world problems" [American Mathematical Society 2019a].

Lamport's works are also characterized by the use of amusing metaphors and associated parables to explain new solutions to problems. Lamport adopted this

approach from another prominent computer scientist of that era, Edsger Dijkstra, who popularized a multiprocess synchronization problem by casting it in terms of philosophers competing for dining utensils. On his “My Writings” page [[Lamport 2019](#)], Lamport writes, “The popularity of the dining philosophers problem taught me that the best way to attract attention to a problem is to present it in terms of a story.” As noted above, Lamport used a bakery metaphor for his solution of Dijkstra’s mutual exclusion problem. To explain the challenge of coordinating multiple bug-prone computers, he used the metaphor of an attack by Byzantine generals. And his solution for fault-tolerant state replication was explained using the story of the island of Paxos and its imaginary part-time parliament.

The chapters of Part I are described below. They are organized roughly in chronological order of the literature they cover. The last chapter in Part I describes Lamport’s professional biography. It provides the context in which his pioneering work arose and sheds light on the people involved with breaking the new ground.

This book would not be complete without a glimpse onto the works themselves. A small selection of the original papers introducing the key notions discussed in the contributed chapters is given in Part II. Yet within the scope of a single book we cannot hope to cover all of Lamport’s important contributions. We therefore encourage readers to visit Lamport’s “My Writings” page [[Lamport 2019](#)], where he gives a complete list of his papers accompanied by historical notes that describe the motivation and context of each result.

Chapter 1: Shared Memory and the Bakery Algorithm

Lamport’s influential works from the 1970s and 1980s came at a time when the fundamental issues of concurrent programming were not well understood. Chapter 1, by Hagit Attiya and Jennifer L. Welch, starts with the seminal Bakery algorithm [[Lamport 1974a](#)] for solving “mutual exclusion,” a fundamental requirement in multiprocessor programming. After developing the Bakery algorithm, Lamport used it as a vehicle for formalizing a number of concepts that facilitate reasoning about concurrent programs. The chapter further covers related concepts that Lamport developed while studying the Bakery algorithm.

At the time, it was known that correct execution may require parallel activities to exclude one another during “critical sections” when they manipulate the same data, in order to prevent undesired interleaving of operations. The mutual exclusion problem originated from Edsger Dijkstra’s pioneering work, which includes

his solution [Dijkstra 1965]. Dijkstra's algorithm, while correct, depends on shared memory accesses being atomic—that one processor reading when another is writing will be made to wait, rather than returning a possibly garbled value. In a sense, it constructs a high-level solution out of the low-level mutual exclusion already implemented by the hardware.

Lamport's Bakery algorithm does not depend on low-level mutual exclusion. In particular, when one processor reads data from a shared variable while the same variable is being updated by another processor, it is acceptable for the former to read garbage, and the algorithm still works! The Bakery algorithm has become textbook material, and most undergraduates in computer science encounter it in the course of their studies.

Among the conceptual contributions emanating from the study of the Bakery algorithm is the notion that processes can make progress independently of the speed of other processes. Rather than preassign turns to processes in a rotation, the Bakery algorithm assigns turns to processes in the order of their arrival. Using the bakery analogy, preassigning turns would be akin to arriving to an empty bakery and being asked to wait for a customer who hasn't even arrived at the bakery yet. Independent progress is a crucial concept that has been used in the design of many subsequent algorithms and in memory architectures. *Wait-freedom*, a condition requiring independent progress despite failures, has its clear roots in this notion and the Bakery doorway concept. It was later extensively explored by Herlihy [1991] and others.

The Bakery algorithm also led Lamport to wonder about the precise semantics of memory when multiple processes interact with shared data. The result is the abstractions of atomic, regular, and safe registers [Lamport 1986c, 1986d].

A register is basically a shared memory location that can be read (loaded) and written (stored) by multiple processes concurrently. Lamport's theory gives each operation on a shared register an explicit duration, starting with an invocation and ending with a result. The registers can be implemented by a variety of techniques, such as replication of the register's data to tolerate faults. Nevertheless, the interactions of processes with an atomic register are supposed to “look like” serial accesses to actual shared memory. The theory also includes semantics of interaction weaker than atomicity, namely, those of regular and safe registers. A regular register captures situations in which processes read different replicas of the register while it is being updated. At any moment in time, some replicas may be updated while others are not, and eventually, all replicas will hold the updated value. The even weaker notion of safe registers allows reads that overlap a write to obtain an arbitrary value. Importantly, these weak semantics suffice

for achieving mutual exclusion: the Bakery algorithm works correctly with safe registers.

It is worth noting that Lamport's study of atomic objects covered in Chapter 1 was restricted to registers, which support only read and write operations. The notion of atomicity was generalized to other data types by [Herlihy and Wing \[1990\]](#), and their term *linearizability* became synonymous with it.

Before the theory of shared registers was completed, Lamport worked on a condition for coherent cache behavior in multiprocessors. That work brought some order to the chaos in this field by introducing sequential consistency [[Lamport 1979b](#)], the last concept covered in the chapter. This simple and intuitive notion provides just the right level of "atomicity" to allow software to work and has become the gold standard for memory consistency models. Today, we design hardware systems with timestamp ordering or partial-store ordering, with added memory fence instructions that allow programmers to make the hardware appear sequentially consistent. Sequential consistency underlies the memory consistency models defined for programming languages like Java and C++. Thus our multicore software runs based on principles described by Leslie Lamport in 1979.

In addition, essentially all nonrelational storage systems developed by companies like Amazon, Google, and Facebook adopt linearizability and sequential consistency as their data coherence guarantees.

Chapter 2: The Notions of Time and Global State in a Distributed System

A prominent type of concurrent system is a distributed one, where processes use messages to interact with each other. Chapter 2, by Karolos Antoniadis and Rachid Guerraoui, covers powerful notions introduced by Lamport that have shaped the way we think about distributed systems as well as the engineering practices of the field.

The first of these is "logical clocks" and the corresponding "logical timestamps," which, in fact, are often referred to as "Lamport timestamps." Many people realized that a global notion of time is not natural for a distributed system, but Lamport was the first to formalize a precise alternative. He defined the "happened before" relation on events—a partial order capturing the causality induced by message exchange. Consider, for example, a process in San Francisco that reads the temperature from a sensor, and a process in London that prints the temperature in San Francisco on a screen. If the Californian process sends a message after reading the sensor and this message is received in London before the temperature is

6 Introduction

presented on the screen, then it is possible that the sensor read in San Francisco has “caused” a specific temperature to be printed out in London. Lamport’s logical clocks capture this potential causality, stipulating that the sensor read happened before the temperature was printed.

Logical clocks are defined in the paper “Time, Clocks, and the Ordering of Events in a Distributed System” [Lamport 1978b] (“Time/Clocks”), which has become the most cited of Lamport’s works. The paper won the very first Principles of Distributed Computing Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing) in 2000 and an ACM SIGOPS Hall of Fame Award in 2007.

At the time of the invention, there was no good way to capture the communication delay in distributed systems except by using real time, and hence the work has become so influential. Lamport realized that the communication delay made those systems very different from shared-memory multiprocessors. The insight came when reading a paper on replicated databases [Johnson and Thomas 1975] and realizing that its logical ordering of commands might violate causality.

Originally, Lamport introduced logical time as a tool to synchronize the copies of replicated service, though it became a powerful notion by itself. Today, this ordering of events is widely used for intuitive proofs of concurrent synchronization algorithms.

In order to correctly replicate a service, Lamport introduced in the same “Time/Clocks” paper one of his most significant contributions, the state machine replication (SMR) paradigm. This paradigm abstracts any service as a centralized state machine—a kind of universal computing engine similar to a Turing machine. A state machine has an internal state, and it processes commands in sequence, each resulting in a new internal state and producing a response. Lamport realized that the daunting task of replicating a service over multiple computers can be made remarkably simple if you present the same sequence of input commands to all replicas and they proceed through an identical succession of states. Chapter 2 describes an SMR solution introduced in Lamport [1978b], which uses logical timestamps to replicate an arbitrary state machine in a distributed system that does not suffer from failures. Fault-tolerant solutions came later, and are the focus of Chapter 4.

A related problem is that of consistently reading the state (taking a “snapshot”) of an arbitrary distributed system. If the global system state is constructed by probing all system components, but due to communication delays they are physically probed at different times, how do we know if that picture is consistent? In a joint work with Mani Chandy, Lamport observed that once you define causal order, the

notion of a “consistent global state” naturally follows [Chandy and Lamport 1985]. In a nutshell, if one event happens before (i.e., causally precedes) another, then the global state should not reflect the latter without the former. This notion is also covered in Chapter 2 along with the Chandy and Lamport [1985] algorithm for obtaining such a consistent snapshot. This is such a powerful notion that others later used it in different domains, like networking, self-stabilization, debugging, and distributed systems. The paper received the 2013 ACM SIGOPS Hall of Fame Award and the 2014 Edsger W. Dijkstra Prize in Distributed Computing.

Chapter 3: Byzantine Faults

Before Lamport developed a full solution for fault-tolerant SMR, he addressed a core ingredient, namely, distributed agreement. Chapter 3, by Christian Cachin, describes the work formulating the agreement problem.

The work arose at SRI International, which had previously been called Stanford Research Institute, in the 1970s. Lamport was part of a team that helped NASA design a robust avionics control system. Formal guarantees were an absolute necessity because of the mission-critical nature of the task. Safety had to be assured against the most extreme system malfunction one could imagine. One of the first challenges the team at SRI was asked to undertake was proving the correctness of a cockpit control scheme that NASA had designed. The scheme relied on three computers replicating the computation and using majority voting to mask any single faulty component.

The team’s work resulted in several foundational concepts and insights regarding these stringent types of robust systems. It included a fundamental definition of robustness in this setting, an abstraction of the coordination problem that underlies any replicated system to date, and a surprising revelation on the prerequisites for three computers to safely run a mission-critical cockpit.

In two seminal works published by Lamport, Marshall Pease, and Robert Shostak [Pease et al. 1980, Lamport et al. 1982], the team first identified a somewhat peculiar vulnerability. They posited that “a failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system.” More generally, a malfunctioning component could function in a manner completely inconsistent with its prescribed behavior, and might appear almost malicious.

The new fault model needed a name; thus the Byzantine generals tale was born, and the name *Byzantine failure* was introduced to capture an arbitrary computer

malfuction. Reaching far beyond the mission-critical avionics system for which it was conceived, the Byzantine fault model is still in use for capturing the worst mishaps and security flaws in systems.

In 1980, Pease, Shostak, and Lamport formulated the problem of reaching coordination despite Byzantine failures [Pease et al. 1980], and in 1982 named it *Byzantine agreement* [Lamport et al. 1982]. Their succinct formulation expresses the control coordination task as the problem of agreeing upon an individual bit, starting with potentially different bits input to each component. One can use it repeatedly in order to keep the system coordinated. In the papers, they show that in the system settings for which NASA designed, four computers are needed for a single bit in the face of a single malfunction. Three are not enough, because then a faulty unit may send conflicting values to the other two units and form a different majority with each one. More generally, they showed that $3t + 1$ units are needed in order to overcome t simultaneously faulty components. To prove this, they used a beautiful symmetry argument now known as the *hexagon argument*. This archetypal argument has been subsequently used in additional settings where a malfunctioning unit that sends conflicting information to different parts of the system looks indistinguishable from a symmetrical situation in which the correct and faulty roles are reversed.

The papers additionally demonstrated that $3t + 1$ units are enough, presenting an algorithm that reaches Byzantine agreement among $3t + 1$ units in $t + 1$ synchronous communication rounds. They further showed that if you use digital signatures, $2t + 1$ units are sufficient and necessary.

The Byzantine agreement problem and its solutions have become pinnacles of fault-tolerant systems. Most systems constructed with redundancy use agreement internally for replication and coordination. Lamport himself later used it to develop fault-tolerant SMR, which is the topic of Chapter 4.

The 1980 paper [Pease et al. 1980] was awarded the 2005 Edsger W. Dijkstra Prize in Distributed Computing, and the 1982 paper [Lamport et al. 1982] received the Jean-Claude Laprie Award in Dependable Computing.

Working on synchronous algorithms for Byzantine agreement made Lamport realize that it is necessary to synchronize clocks among the processes. The chapter also includes a brief recollection of another seminal work [Lamport and Melliar-Smith 1985] in which Lamport, together with Michael Melliar-Smith, formalized the Byzantine clock synchronization problem and gave its first solutions.

The last topic covered in the chapter is a “one off” work Lamport did in cryptography, one-time signatures based on one-way functions.

Chapter 4: State Machine Replication with Benign Failures

The first SMR solution Lamport presented in his 1978 “Time/Clocks” paper assumed there are no failures, and it made use of logical time to step replicas through the same command sequence. With the growing understanding of reaching agreement in distributed systems, it was time for Lamport to go back to state machine replication and address failures and lack of synchrony. This is the topic of Chapter 4, by Robbert van Renesse.

In 1989, Lamport designed a fault-tolerant algorithm called Paxos [Lamport 1998a]. Continuing his trend of humorous parable-telling, the paper presents the imaginary story of an ancient parliament on the Greek island of Paxos, where the absence of any number of its members, or possibly all of them, can be tolerated without losing consistency.

Unfortunately, the setting as a Greek parable made the paper difficult for most readers to comprehend, and it took nine years from submission to publication in 1998. But the 1989 DEC technical report did get noticed. Lamport’s colleague Butler Lampson evangelized the idea to the distributed computing community [Lampson 1996].

Paxos stitches together a succession of agreement decisions into a sequence of state machine commands in an optimized manner. Importantly, the first phase of the agreement component given in the Paxos paper (called Synod) can be avoided when the same leader presides over multiple decisions; this first phase needs to be performed only when a leader needs to be replaced. This insightful breakthrough accounts for much of the popularity of Paxos, and was later called Multi-Paxos by a Google team that implemented the algorithm [Chandra et al. 2007]. Lamport’s Paxos paper won the ACM SIGOPS Hall of Fame Award in 2012.

SMR and Paxos have become the de facto standard framework for designing and reasoning about replication methods. State machine replication à la Paxos is now widely offered and deployed as an external service via libraries and toolkits such as Google’s Chubby [Burrows 2006], Apache’s open-source ZooKeeper [Hunt et al. 2010], the popular open-source etcd/raft library, and more. Virtually all companies building critical information systems, including Google, Yahoo, Microsoft, and Amazon, have adopted the Paxos foundations. The engineering of reliable systems led to several important variants and modifications of Paxos. The chapter briefly describes Disk Paxos [Gafni and Lamport 2003], Cheap Paxos [Lamport and Massa 2004], Vertical Paxos [Lamport et al. 2009a], and Stoppable Paxos [Lamport et al. 2010].

Chapter 5: Formal Specification and Verification

Even before he worked on the Bakery algorithm, Lamport learned, through an erroneous manuscript he submitted for publication, the importance of rigorously specifying and proving algorithms correct. Chapter 5, by Stephan Merz, describes his quest for good foundations and tools to describe solutions and prove their correctness.

Lamport has made central contributions to the theory of specification and verification of concurrent programs. He was the first to articulate the notions of safety properties and liveness properties for asynchronous distributed algorithms. These were the generalization of “partial correctness” and “total correctness” properties previously defined for sequential programs. Today, safety and liveness form the standard classification for correctness properties of asynchronous distributed algorithms.

Another work, with Martin Abadi, introduced *prophecy variables*: an abstraction that can be added to an algorithm model in order to handle a situation where an algorithm resolves a nondeterministic choice before its specification does. These complement the previously suggested notion of *history variables*, auxiliary variables that record past actions of the algorithm. [Abadi and Lamport \[1991\]](#) pointed out situations where such problems arise and developed the foundations needed to support this extension to the theory. Moreover, they proved that whenever an algorithm meets a specification, where both are expressed as state machines, the correspondence between them can be proved using a combination of prophecy and history variables. This work won the 2008 LICS Test-of-Time Award.

Lamport realized that computer scientists need more than foundational notions for reasoning about concurrency. They need languages to formally express solutions and tools for verifying their correctness. Chapter 5 gives an overview of the TLA (temporal logic of actions) logic and the specification language TLA⁺ Lamport developed for modeling and verifying distributed algorithms and systems. TLA and TLA⁺ support specification and proof of both safety and liveness properties using notation based on temporal logic.

Lamport has supervised the development of verification tools based on TLA⁺, notably the TLC model checker built by Yuan Yu. TLA⁺ and TLC have been used to describe and analyze real systems. For example, TLA⁺ was used to find a major error in the coherence protocol used in the hardware for Microsoft’s Xbox 360 prior to its release in 2005. They were also used for the analysis of cache coherence protocols at DEC and Intel. To teach engineers how to use his formal specification tools, Lamport wrote a book [2002] and also developed the PlusCAL [2009] formal

language and tools for use in verifying distributed algorithms. At the time of this writing, Leslie Lamport continues to work actively on enhancing and evangelizing the TLA⁺ toolset.

Chapter 6: Biography

Chapter 6, by Roy Levin, tells of Lamport's career. It views his works in historical perspective, providing the context in which they arose during four decades of an evolving industry: from the introduction of the first personal computer to an era when parallel and distributed multiprocessors are abundant. Through this lens, it portrays their long-lasting impact.

The biography also tells of L^AT_EX, perhaps Lamport's most influential contribution outside the field of concurrency. As a prolific author, Lamport would naturally wish for a convenient typesetting tool. He did not just wish for one, he created one for the entire community. Lamport's L^AT_EX system [Lamport 1994b] is a set of macros for use with Donald Knuth's T_EX typesetting system. In creating L^AT_EX, Lamport brought to T_EX three concepts that he borrowed from Brian Reid's Scribe system and comprehensively elaborated:

- The concept of “typesetting environment”
- A strong emphasis on structural rather than typographic markup
- A generic document design, flexible enough to be adequate for a wide variety of documents

L^AT_EX is a system that provides the quality of T_EX and a lot of its flexibility, but is much easier to use. It has became the de facto standard for technical publishing in computer science and many other fields.

Several other works of Lamport's that are omitted from the technical chapters of this book surface in the biography chapter. The chapter underscores a lifetime achievement and the long-lasting impact Lamport has had on the computer science field. It paints a picture of how his timeless impact on the foundations of concurrency formed. It threads Lamport's thought process as he developed solutions for the challenges he tackled.

The biography chapter gives a glimpse into the professional interactions, and even some conflicts, that the pioneers of the concurrency arena had as they broke new grounds. The chapter concludes by giving a voice to the people behind the achievements, notably Lamport himself and additionally the colleagues around him, who have inspired, collaborated, and helped him drive worldwide impact.

Closing Remarks

If one could travel back in time to 1974, perhaps one would find Leslie Lamport arranging a queue for customers at his busy local neighborhood bakery to be served one at a time via the Bakery algorithm. This and Lamport's other pioneering works—many with amusing names and associated parables—have become pillars of computer science. His collection forms the foundation of broad areas in concurrency and has influenced the specification, development, and verification of concurrent systems. Any time you access a modern computer, you are likely to be impacted by Leslie Lamport's algorithms.

This book touches on a lifetime of contributions by Leslie Lamport to the field of concurrency and on the extensive influence he has had on people working in the field. Those who have collaborated with him have often found the experience remarkable and sometimes even career-altering.

And all of this work started with the quest to understand how to organize a queue at the local bakery.

The Computer Science of Concurrency: The Early Years

Leslie Lamport (Microsoft Research)

28 February 2015

To Edsger Dijkstra

It is insufficiently considered that men more often require to be reminded than informed.

—Samuel Johnson

1 Foreword

I don't know if concurrency is a science, but it is a field of computer science. What I call *concurrency* has gone by many names, including parallel computing, concurrent programming, and multiprogramming. I regard distributed computing to be part of the more general topic of concurrency. I also use the name *algorithm* for what were once usually called programs and were generally written in pseudo-code.

This is a personal view of the first dozen years of the history of the field of concurrency—a view from today, based on 40 years of hindsight. It reflects my biased perspective, so despite covering only the very beginning of what was then an esoteric field, it is far from complete. The geneses of my own contributions are described in comments in my publications web page.

The omission that would have seemed most striking to someone reading this history in 1977 is the absence of any discussion of programming languages. In the late 1960s and early 1970s, most papers considered to be about concurrency were about language constructs for concurrent programs. A problem such as mutual

exclusion was considered to be solved by introducing a language construct that made its solution trivial. This article is not about concurrent programming; it is about concurrent algorithms and their underlying principles.

2 The Beginning: Mutual Exclusion

2.1 The Problem

While concurrent program execution had been considered for years, the computer science of concurrency began with Edsger Dijkstra's seminal 1965 paper that introduced the mutual exclusion problem [5]. He posed the problem of synchronizing N processes, each with a section of code called its *critical section*, so that the following properties are satisfied:

Mutual Exclusion No two critical sections are executed concurrently. (Like many problems in concurrency, the goal of mutual exclusion is to eliminate concurrency, allowing us to at least pretend that everything happens sequentially.)

Livelock Freedom If some process is waiting to execute its critical section, then some process will eventually execute its critical section.

Mutual exclusion is an example of what is now called a *safety* property, and livelock freedom is called a *liveness* property. Intuitively, a safety property asserts that something bad never happens; a liveness property asserts that something good must eventually happen. Safety and liveness were defined formally in 1985 [1].

Dijkstra required a solution to allow any computer to halt outside its critical section and associated synchronizing code. This is a crucial requirement that rules out simple, uninteresting solutions—for example, ones in which processes take turns entering their critical sections. The 1-buffer case of the producer-consumer synchronization algorithm given below essentially is such a solution for $N = 2$.

Dijkstra also permitted no real-time assumption. The only progress property that could be assumed was *process fairness*, which requires every process that hasn't halted to eventually take a step. In those days, concurrency was obtained by having multiple processes share a single processor. One process could execute thousands of steps while all other processes did nothing. Process fairness was all one could reasonably assume.

Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented

as a sequence of states, where a state consists of an assignment of values to the algorithm's variables plus other necessary information such as the control state of each process (what code it will execute next). I have found this to be the most generally useful model of computation—for example, it underlies a Turing machine. I like to call it the *standard model*.

The need for careful proofs should have become evident a few months later, when the second published mutual exclusion algorithm [9] was shown to be incorrect [10]. However, incorrect concurrent algorithms are still being published and will no doubt continue to be for a long time, despite modern tools for catching errors that require little effort—in particular, model checkers.

2.2 The First “Real” Solution

Although of little if any practical use, the bakery algorithm [11] has become a popular example of a mutual exclusion algorithm. It is based on a protocol sometimes used in retail shops: customers take successively numbered tickets from a machine, and the lowest-numbered waiting customer is served next. A literal implementation of this approach would require a ticket-machine process that never halts, violating Dijkstra's requirements. Instead, an entering process computes its own ticket number by reading the numbers of all other synchronizing processes and choosing a number greater than any that it sees.

A problem with this algorithm is that ticket numbers can grow without bound. This shouldn't be a practical problem. If each process chooses a number at most one greater than one that was previously chosen, then numbers should remain well below 2^{128} . However, a ticket number might have to occupy more than one memory word, and it was generally assumed that a process could atomically read or write at most one word.

The proof of correctness of the algorithm revealed that the read or write of an entire number need not be atomic. The bakery algorithm is correct as long as reading a number returns the correct value if the number is not concurrently being written. It doesn't matter what value is returned by a read that overlaps a write. The algorithm is correct even if reading a number while it is changing from 9 to 10 obtains the value 2496.

This amazing property of the bakery algorithm means that it implements mutual exclusion without assuming that processes have mutually exclusive access to their ticket numbers. It was the first algorithm to implement mutual exclusion without assuming any lower-level mutual exclusion. In 1973, this was considered impossible [4, page 88]. Even in 1990, experts still thought it was impossible [21, question 28].

One problem remained: How can we maintain a reasonable bound on the values of ticket numbers if a read concurrent with a write could obtain any value? For example, what if reading a number while it changes from 9 to 10 can obtain the value 2^{2496} ? A closely related problem is to implement a system clock that provides the current time in nanoseconds if reads and writes of only a single byte are atomic, where a read must return a time that was correct sometime during the read operation. Even trickier is to implement a cyclic clock. I recommend these problems as challenging exercises. Solutions have been published [12].

2.3 A Rigorous Proof of Mutual Exclusion

Previous correctness proofs were based on the standard model, in which an execution is represented as a sequence of states. This model assumes atomic transitions between states, so it doesn't provide a natural model of the bakery algorithm with its non-atomic reads and writes of numbers.

Before I discuss a more suitable model, consider the following conundrum. A fundamental problem of interprocess synchronization is to ensure that an operation executed by one process precedes an operation executed by another process. For example, mutual exclusion requires that if two processes both execute their critical sections, then one of those operation executions precedes the other. Many modern multiprocessor computers provide a Memory Barrier (MB) instruction for implementing interprocess synchronization. Executing an instruction *A* then an MB then instruction *B* in a single process ensures that the execution of *A* precedes that of *B*. Here is the puzzle: An MB instruction enforces an ordering of two operations performed by the same process. Why is that useful for implementing interprocess synchronization, which requires ordering operations performed by different processes? The reader should contemplate this puzzle before reading the following description of the *two-arrow* model.

In the two-arrow model, an execution of the algorithm is represented by a set of *operation executions* that are considered to have a finite duration with starting and stopping times. The relations \rightarrow and \rightarrowtail on this set are defined as follows, for arbitrary operation executions *A* and *B*:

$A \rightarrow B$ is true iff (if and only if) *A* ends before *B* begins.

$A \rightarrowtail B$ is true iff *A* begins before *B* ends.

It is easy to check that these relations satisfy the following properties, for any operation executions *A*, *B*, *C*, and *D*:

1. (a) $A \rightarrow B \rightarrow C$ implies $A \rightarrow C$ (\rightarrow transitively closed)
(b) $A \not\rightarrow A$. (\rightarrow irreflexive)

2. $A \rightarrow B$ implies $A \rightarrowtail B$ and $B \not\rightarrowtail A$.
3. $A \rightarrow B \rightarrowtail C$ or $A \rightarrowtail B \rightarrow C$ implies $A \rightarrowtail C$.
4. $A \rightarrow B \rightarrowtail C \rightarrow D$ implies $A \rightarrow D$.

The model abstracts away the explicit concept of time and assumes only a set of operation executions and relations \rightarrow and \rightarrowtail on it satisfying A1–A4. (An additional property is needed to reason about liveness, which I ignore here.)

Proving correctness of the bakery algorithm requires some additional assumptions:

- All the operation executions within a single process are totally ordered by \rightarrow .
- For any read R and write W of the same variable, either $R \rightarrowtail W$ or $W \rightarrow R$ holds.

Each variable in the algorithm is written by only a single process, so all writes to that variable are ordered by \rightarrow . We assume that a read that doesn't overlap a write obtains the correct value. More precisely, if a read R of a variable satisfies $R \rightarrow W$ or $W \rightarrow R$ for every write W of the variable, then R obtains the value written by the latest write W with $W \rightarrow R$.

With these assumptions, the two-arrow formalism provides the most elegant proof of the bakery algorithm that I know of. Such a proof of a variant of the algorithm appears in [14].

The conundrum of the MB command described at the beginning of this section is easily explained in terms of the two-arrow formalism. Suppose we want to ensure that an operation execution A in process p precedes an operation execution D in a different process q —that is, to ensure $A \rightarrow D$. Interprocess communication by accessing shared registers can reveal only that an operation execution C in q sees the effect of an operation execution B in p , which implies $B \rightarrowtail C$. The only way to deduce a \rightarrow relation from a \rightarrowtail relation is with A4. It allows us to deduce $A \rightarrow D$ from $B \rightarrowtail C$ if $A \rightarrow B$ and $C \rightarrow D$. The latter two \rightarrow relations can be ensured by using MB instructions, which enforces \rightarrow relations between operation executions by the same process.

3 Producer-Consumer Synchronization

3.1 The FIFO Queue

The second fundamental concurrent programming problem to be studied was producer-consumer synchronization. This form of synchronization was used at the

```
--algorithm PC {
  variables in = Input, out = < >, buf = < >;
  fair process (Producer = 0) {
    P: while (TRUE) {
      await Len(buf) < N ;
      buf := Append(buf, Head(in)) ;
      in := Tail(in) } }

  fair process (Consumer = 1) {
    C: while (TRUE) {
      await Len(buf) > 0 ;
      out := Append(out, Head(buf)) ;
      buf := Tail(buf) } } }
```

Figure 1 Producer-consumer synchronization.

hardware level in the earliest computers, but it was first identified as a concurrency problem by Dijkstra in 1965, though not published in this formulation until 1968 [6]. Here, I consider an equivalent form of the problem: a bounded FIFO (first-in-first-out) queue. It can be described as an algorithm that reads inputs into an N -element buffer and then outputs them. The algorithm uses three variables:

in The infinite sequence of unread input values.

buf A buffer that can hold up to N values.

out The sequence of values output so far.

A *Producer* process moves values from *in* to *buf*, and a *Consumer* process moves them from *buf* to *out*. In 1965 the algorithm would have been written in pseudo-code. Today, we can write it in the PlusCal algorithm language [15] as algorithm *PC* of Figure 1. The initial value of the variable *in* is the constant *Input*, which is assumed to be an infinite sequence of values; variables *buf* and *out* initially equal the empty sequence. The processes *Producer* and *Consumer* are given the identifiers 0 and 1. In PlusCal, an operation execution consists of execution of the code from one label to the next. Hence, the entire body of each process's **while** loop is executed atomically. The **await** statements assert enabling conditions of the actions. The keywords **fair** specify process fairness.

Figure 2 shows the first four states of an execution of the algorithm represented in the standard model. The letter *P* or *C* atop an arrow indicates which process's atomic step is executed to reach the next state.

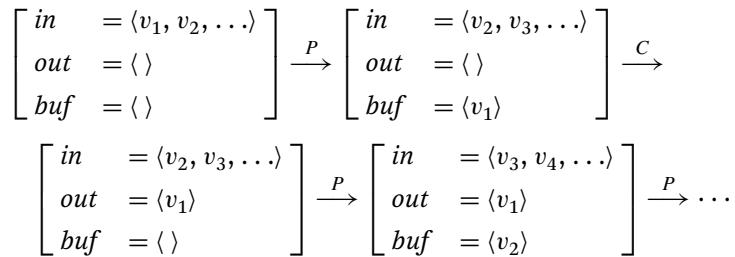


Figure 2 An execution of the FIFO queue.

Algorithm *PC* is a specification; a bounded FIFO queue must implement that specification. A specification is a definition, and it makes no formal sense to ask if a definition is correct. However, we can gain confidence that this algorithm does specify a bounded FIFO queue by proving properties of it. The most important class of properties one proves about an algorithm are invariance properties. A state predicate is an *invariant* iff it is true in every state of every execution. The following invariant of algorithm *PC* suggests that it is a correct specification of an *N*-element bounded queue:

$$(Len(buf) \leq N) \wedge (Input = out \circ buf \circ in)$$

where *Len(buf)* is the length of the sequence *buf* and \circ is sequence concatenation.

The basic method for proving that a predicate *Inv* is an invariant of a concurrent algorithm was introduced by Edward Ashcroft in 1975 [2]. We find a suitable predicate *I* (the inductive invariant) and prove that (i) *I* is true in every initial state, (ii) *I* is left true by every step of the algorithm, and (iii) *I* implies *Inv*. It is easy to prove that the state predicate above is an invariant of algorithm *PC*. The appropriate inductive invariant *I* is the conjunction of this invariant with a predicate asserting that each variable has a “type-correct” value. (PlusCal is an untyped language.)

3.2 Another Way of Looking at a FIFO Queue

The FIFO queue specification allows only a single initial state, and executing either process’s action can produce only a single next state. Hence the execution of Figure 2 is completely determined by the sequence $P \rightarrow C \rightarrow P \rightarrow P \rightarrow \dots$ of atomic-action executions. For $N = 3$, all such sequences are described by the graph in Figure 3. The nodes of the graph are called *events*, each event representing an

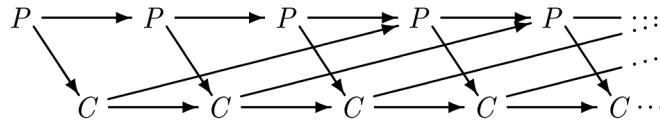


Figure 3 An event history for the FIFO queue with $N = 3$.

atomic execution of the algorithm step with which the event is labeled. The graph defines an irreflexive partial order \prec on the events, where $e \prec f$ iff $e \neq f$ and there is a path through the graph from event e to event f . For want of a standard term for it, I will call such a partially ordered set of events, in which events are labeled with atomic steps, an *event history*.

This event history describes all sequences of states that represent executions of algorithm PC in the standard model. Such a sequence of states is described by a sequence of infinitely many P and C events—that is, by a total ordering of the events in the event history. A total ordering of these events describes a possible execution of algorithm PC iff it is consistent with the partial order \prec . To see this, observe that the downward pointing diagonal arrows imply that the i^{th} P event (which moves the i^{th} input to the buffer) must precede the i^{th} C event (which moves that input from the buffer to the output). The upward pointing diagonal arrows indicate that the i^{th} C event must precede the $(i + 3)^{\text{rd}}$ P event, which is necessary to ensure that there is room for the $(i + 3)^{\text{rd}}$ input value in the buffer, which can hold at most 3 elements.

We can view the event history of the figure to be the single “real” execution of algorithm PC . The infinitely many different executions in the standard model are artifacts of the model; they are not inherently different. Two events not ordered by the \prec relation—for example, the second C event and the fourth P event—represent operations that can be executed concurrently. However, the standard model requires concurrent executions of the two operations to be modeled as occurring in some order.

3.3 Mutual Exclusion versus Producer-Consumer Synchronization

Producer-consumer synchronization is inherently deterministic. On the other hand, mutual exclusion synchronization is inherently nondeterministic. It has an inherent race condition: two processes can compete to enter the critical section, and either might win.

Resolving a race requires an *arbiter*, a device that decides which of two events happens first [3]. An arbiter can take arbitrarily long to make its decision. (A well-

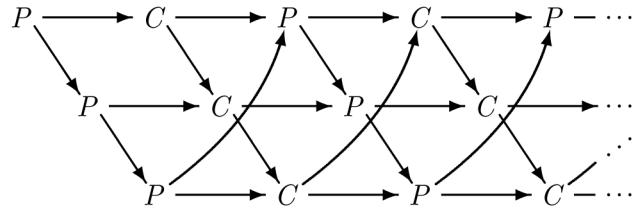


Figure 4 Another view of the FIFO queue for $N = 3$.

designed arbiter has an infinitesimal probability of taking very long.) Any mutual exclusion algorithm can therefore, in principle, take arbitrarily long to allow some waiting process to enter its critical section. This is not an artifact of any model. It appears to be a law of nature.

Producer-consumer synchronization has no inherent nondeterminism, hence no race condition. It can be implemented without an arbiter, so each operation can be executed in bounded time. It is a fundamentally different class of problem than mutual exclusion.

3.4 The FIFO Queue as an N -Process System

The graph in Figure 3 is drawn with two rows, each containing the events corresponding to actions of one of the two processes. Figure 4 is the same graph drawn with three rows. We can consider the three rows to be three separate processes. If we number these rows 0, 1, and 2 and we number the elements in the *Input* sequence starting from 0, then the events corresponding to the reading and outputting of element i of *Input* are in row $i \bmod 3$. We can consider each of those rows to be a process, making the FIFO queue a 3-process system for $N = 3$, and an N -process system in general. If we were to implement the variable *buf* with an N -element cyclic buffer, each of these processes would correspond to a separate buffer element.

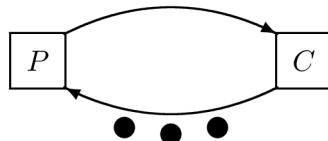
In the event history model, any totally ordered subset of events can be considered a process. The standard model has no inherent notion of processes. In that model, an execution is just a sequence of states. Processes are an artifact of the way the sequence of states is represented. The set of executions of algorithm *PC* can also be described by an N -process PlusCal algorithm.

3.5 Generalized Producer-Consumer Synchronization

The generalization of producer-consumer synchronization is marked-graph synchronization. Marked graphs were introduced by Holt and Commoner in 1970 [8].

A marked graph is a directed graph together with a *marking* that assigns a finite set of indistinguishable tokens to each arc. A node is *fired* in a marking by removing one token from each of its input arcs and adding one token to each of its output arcs (producing a new marking). A *firing sequence* of a marked graph is a sequence of firings that can end only with a marking in which no node may be fired. (By definition of firing, a node can be fired iff it has at least one token on each input arc.)

A marked graph synchronization problem is described by labeling the nodes of a marked graph with the names of atomic operations. This specifies that a sequence of atomic operation executions is permitted iff it is the sequence of labels of the nodes in a possible firing sequence of the marked graph. For example, the following marked graph describes the FIFO queue for $N = 3$.



A token on the top arc represents a value in the buffer, and a token on the bottom arc represents space for one value in the buffer. Observe that the number of tokens on this marked graph remains constant throughout a firing sequence. The generalization of this observation to arbitrary marked graphs is that the number of tokens on any cycle remains constant.

All executions of a marked graph synchronization algorithm are described by a single event history. Marked graph synchronization can be implemented without an arbiter, so each operation can be executed in a bounded length of time.

Marked graphs can be viewed as a special class of Petri nets [18]. Petri nets are a model of concurrent computation especially well-suited for expressing the need for arbitration. Although simple and elegant, Petri nets are not expressive enough to formally describe most interesting concurrent algorithms. Petri nets have been used successfully to model some aspects of real systems, and they have been generalized to more expressive languages. But to my knowledge, neither Petri nets nor their generalizations have significantly influenced the field of concurrent algorithms.

3.6 The Two-Arrow Formalism Revisited

Let \mathcal{E} be an event history with partial order \prec . Suppose we partition \mathcal{E} into nonempty disjoint subsets called *operation executions*. We can define two relations \longrightarrow and \dashrightarrow on the set of operation executions as follows, for any operation executions A

and B :

$$A \longrightarrow B \text{ iff } \forall e \in A, f \in B : e \prec f.$$

$$A \dashrightarrow B \text{ iff } \exists e \in A, f \in B : e \prec f.$$

It is straightforward to see that these definitions (and the assumption that \prec is an irreflexive partial order) imply properties A1–A4 of Section 2.3. Thus, we can obtain a two-arrow representation of the execution of an algorithm with non-atomic operations from an event history whose events are the atomic events that make up the operation executions. The event history does not have to be discrete. Its events could be points in a space-time continuum, where \prec is the causality relation introduced by Minkowski [17].

4 Distributed Algorithms

Pictures of event histories were first used to describe distributed systems. Figure 5 is an event history that appeared as an illustration in [13]. The events come from three processes, with time moving upwards. A diagonal arc joining events from two different processes represents the causality relation requiring that a message must be sent before it is received. For example, the arc from q_4 to r_3 indicates that event q_4 of the second process sent a message that was received by event r_3 of the third process.

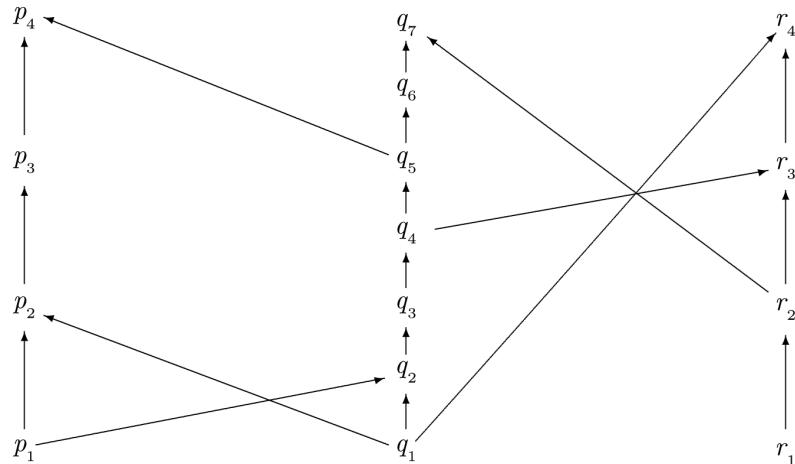


Figure 5 An event history for a distributed system.

In general, executions of such a distributed system can produce different event histories. For example, in addition to the history of Figure 5, there might be an event history in which the message sent by event q_1 is received before the message sent by event q_4 . In such a case, there is true nondeterminism and the system requires arbitration.

Let a *consistent* cut of an event history consist of a set C of events such that for every two events c and d , if event c is in C and $d \prec c$, then d is in C . For example, $\{p_1, q_1, q_2, r_1, r_2\}$ is a consistent cut of the event history of Figure 5. Every consistent cut defines a global state of the system during some execution in the standard model—the state after executing the steps associated with the events in the consistent cut.

An event history like that of Figure 5 allows incompatible consistent cuts—that is two consistent cuts, neither of which is a subset of the other. They describe possible global states that, in the standard model, may not occur in the same execution. This shows that there is no meaningful concept of a unique global state at an instant. For example, there are different consistent cuts containing only events q_1 and q_2 of the second process. They represent different possible global states immediately after the process has executed event q_2 . There is no reason to distinguish any of those global states as *the* global state at that instant.

Because the standard model refers to global states, it has been argued that the model should not be used for reasoning about distributed algorithms and systems. While this argument sounds plausible, it is wrong. An invariant of a global system is a meaningful concept because it is a state predicate that is true for all possible global states, and so does not depend on any preferred global states. The problem of implementing a distributed system can often be viewed as that of maintaining a global invariant even though different processes may have incompatible views of what the current state is at any instant.

Thinking is useful, and multiple ways of thinking can be even more useful. However, while event histories may be especially useful for helping us understand distributed systems, the best way to reason about these systems is usually in terms of global invariants. The standard model provides the most practical way to reason about invariance.

5 Afterwards

After distributed systems, the next major step in concurrent algorithms was the study of fault tolerance. The first scientific examination of fault tolerance was Dijkstra's seminal 1974 paper on self-stabilization [7]. However, as sometimes

happens with work that is ahead of its time, that paper received little attention and was essentially forgotten for a decade. A survey of fault tolerance published in 1978 [20] does not mention a single algorithm, showing that fault tolerance was still the province of computer engineering, not of computer science.

At about the same time that the study of fault-tolerant algorithms began in earnest, the study of models of concurrency blossomed. Arguably, the most influential of this work was Milner's CCS [16]. These models were generally event-based, and avoided the use of state. They did not easily describe algorithms or the usual way of thinking about them based on the standard model. As a result, the study of concurrent algorithms and the study of formal models of concurrency split into two fields. A number of formalisms based on the standard model were introduced for describing and reasoning about concurrent algorithms. Notable among them is temporal logic, introduced by Amir Pnueli in 1977 [19].

The ensuing decades have seen a huge growth of interest in concurrency—particularly in distributed systems. Looking back at the origins of the field, what stands out is the fundamental role played by Edsger Dijkstra, to whom this history is dedicated.

References

- [1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
- [2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
- [3] J. C. Barros and B. W. Johnson. Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay. *IEEE Transactions on Computers*, C-32(7):603–614, July 1983.
- [4] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.
- [5] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
- [6] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968. Originally appeared as EWD123 (1965).
- [7] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
- [8] A. Holt and F. Commoner. Events and conditions. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 3–52. Project MAC, June 1970.
- [9] Harris Hyman. Comments on a problem in concurrent programming control. *Communications of the ACM*, 9(1):45, January 1966.

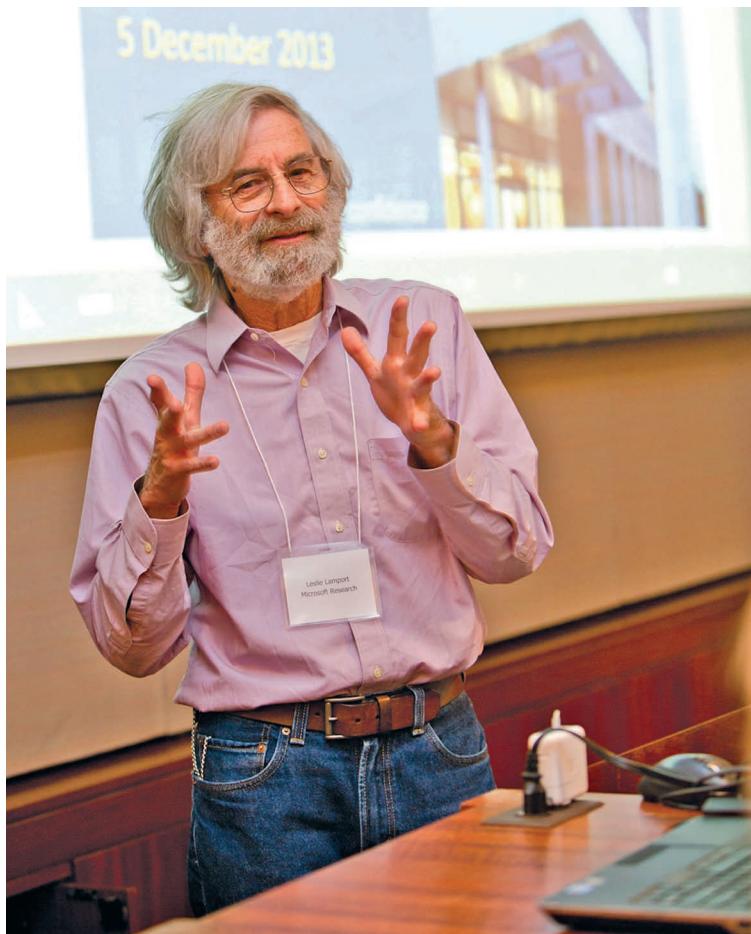
- [10] D. E. Knuth. Additional comments on a problem in concurrent program control. *Communications of the ACM*, 9(5):321–322, May 1966.
- [11] Leslie Lamport. A new solution of Dijkstra’s concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
- [12] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.
- [13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [14] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.
- [15] Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing, ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer-Verlag, 2009.
- [16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.
- [17] H. Minkowski. Space and time. In *The Principle of Relativity*, pages 73–91. Dover, 1952.
- [18] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In Cicely M. Popplewell, editor, *Information Processing 1962, Proceedings of IFIP Congress 62*, pages 386–390. North-Holland, 1962.
- [19] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.
- [20] B. Randell, P. A. Lee, and P.C. Treleaven. Reliability issues in computing system design. *Computing Surveys*, 10(2):123–165, June 1978.
- [21] Brian A. Rudolph. Self-assessment procedure xxi. *Communications of the ACM*, 33(5):563–575, May 1990.

PART

TECHNICAL PERSPECTIVES ON LAMPORT'S WORK



Leslie Lamport skiing in Norway in 2002. (Photo by Dag Johansen)



Leslie Lamport lecturing at Cornell University, Ithaca, NY, December 2, 2013.

Shared Memory and the Bakery Algorithm

Hagit Attiya, Jennifer L. Welch

1.1

Introduction

Starting in the early 1970s, Lamport was intrigued by the question of how to achieve synchronization and exclusion for multiprocessor computer systems in software, even when the hardware does not provide such facilities. This study of shared memory marked Lamport’s first venture into distributed algorithms [Lamport 2019].

A multiprocessor consists of a set of sequential processes that communicate with each other through shared memory modules, which support fetch and store requests from the processes to read and write shared memory registers. *Mutual exclusion* is a fundamental problem in multiprocessors in which each process may occasionally request exclusive access to some resource for some finite period of time. A solution to the problem should ensure that each request is eventually granted.

Previous algorithms proposed for this problem suffered from several drawbacks that Lamport noticed. One drawback was that the algorithms were not fault tolerant: if a process crashed at an inopportune time, then the entire system would halt. Lamport [1974a] presented the Bakery algorithm for mutual exclusion, which avoids this problem. We present this algorithm in Section 1.2.

Another drawback was that previous work had assumed the shared memory was atomic, meaning that the reads and writes appeared to occur instantaneously and in some sequential order. But this is a strong assumption. A remarkable aspect of the Bakery algorithm is that its correctness does not depend on this assumption! It is still correct even if the shared memory satisfies much weaker properties.

To capture the weaker types of shared memory, Lamport introduced a new formalism [Lamport 1986c, 1986d]. Unlike most prior work, this formalism is not

based on atomic actions. Instead, a system execution is modeled as a set of operation executions together with two precedence relations on the operation executions: “precedes” and “can affect”. Different types of shared memory are captured by increasingly stronger sets of axioms these relations must satisfy. This formalism is then used to show how various types of shared memory can be simulated by one another. We present these simulations and their informal proofs in Section 1.3 and an overview of the formalism and its applications in Section 1.4.

Even earlier, Lamport introduced a shared memory consistency condition called *sequential consistency* [Lamport 1979b]. Although a similar notion was used previously [Dijkstra 1971, Lamport 1977a, Owicky and Gries 1976], this paper was the first to coin the term *sequential consistency*. The paper presents necessary and sufficient conditions for implementing sequential consistency in a multiprocessor. We discuss these results in Section 1.5.

1.2

Flavors of the Bakery Algorithm

The section describes the *Bakery algorithm* [Lamport 1974a], a mutual exclusion algorithm, as a vehicle to explain different types of shared memory. Unlike prior mutual exclusion algorithms, the Bakery algorithm tolerates crash failures of participating processes.

We first describe the algorithm assuming intuitive, atomic registers are used. We prove its correctness. Then we introduce weaker forms of shared memory, namely, regular and safe registers, and argue that the Bakery algorithm is still correct. For safe registers, shared variables used in the algorithm can increase arbitrarily, but we describe a fix for this undesirable behavior [Lamport 1977].

1.2.1 The Mutual Exclusion Problem

The *mutual exclusion problem* concerns a group of processes that occasionally need access to some resource that cannot be used simultaneously by more than a single process; for example, some output device. Each process may need to execute a code segment called a *critical section* such that, informally speaking, at any time, at most one process is in the critical section (*mutual exclusion*), and if one or more processes try to enter the critical section, then one of them eventually succeeds as long as no process stays in the critical section forever (*no deadlock*).

The above properties do not provide any guarantee on an individual basis because a process may try to enter the critical section and yet fail because it is always bypassed by other processes. A stronger property, which implies no deadlock, is

no lockout: If a process wishes to enter the critical section, then it will eventually succeed as long as no process stays in the critical section forever. (This property is sometimes called *no starvation*.) Later we will see an even stronger property that limits the number of times a process might be bypassed while trying to enter the critical section.

Original solutions to the mutual exclusion problem relied on special synchronization support, such as semaphores and monitors. Here we focus on *distributed* software solutions using ordinary shared variables.

Each process executes some additional code before and after the critical section to ensure the above properties; we assume the program of a process is partitioned into the following sections:

Entry (trying). The code executed in preparation for entering the critical section.

Critical. The code to be protected from concurrent execution.

Exit. The code executed on leaving the critical section.

Remainder. The rest of the code.

Each process cycles through these sections in the order remainder, entry, critical, and exit. If a process wants to enter the critical section, it first executes the entry section; after that, the process enters the critical section; then the process releases the critical section by executing the exit section and returning to the remainder section.

A mutual exclusion algorithm consists of code for the entry and exit sections and should work no matter what goes in the critical and remainder sections. In particular, a process may transition from the remainder section to the entry section any number of times, either finite or infinite. We assume that the variables, both shared and local, accessed in the entry and exit sections are *not* accessed in the critical and remainder sections. We also assume that no process stays in the critical section forever.

To capture these requirements, we make the following assumptions in the formal model. If a process takes a step while in the remainder (resp., critical) section, it immediately enters the entry (resp., exit) section. The definition of admissible execution is changed to allow a process to stop in the remainder section. Thus an execution is *admissible* if for every process p_i , either p_i takes an infinite number of steps or p_i ends in the remainder section.

More formally, an algorithm for a shared memory system solves the mutual exclusion problem with no deadlock (or no lockout) if the following hold:

Mutual exclusion. In every configuration of every execution, at most one process is in the critical section.

No deadlock. In every admissible execution, if some process is in the entry section in a configuration, then there is a later configuration in which some process is in the critical section.

No lockout. In every admissible execution, if some process is in the entry section in a configuration, then there is a later configuration in which *that same* process is in the critical section.

We also require that in an admissible execution, no process is ever stuck in the exit section; this is called the *unobstructed exit* condition. In all the algorithms presented in this chapter, the exit sections are straight-line code (i.e., no loops), and thus the condition obviously holds.

Note that the mutual exclusion condition is required to hold in every execution, not just admissible ones.

1.2.2 The Bakery Algorithm

In this section, we describe the Bakery algorithm for mutual exclusion among n processes; the algorithm provides mutual exclusion and no lockout.

The main idea is to consider processes wishing to enter the critical section as customers in a bakery. Each customer arriving at the bakery gets a number, and the one with the smallest number is the next to be served. The number of a customer who is not standing in line is 0 (which does not count as the smallest ticket).

To make the bakery metaphor more concrete, we employ the following shared data structures: *Number* is an array of n integers that holds in its i th entry the number of p_i ; *Choosing* is an array of n Boolean values such that *Choosing*[i] is true while p_i is in the process of obtaining its number.

Each process p_i wishing to enter the critical section tries to choose a number that is greater than all the numbers of the other processes and writes it to *Number*[i]. This is done by reading *Number*[0], ..., *Number*[$n - 1$] and taking the maximum among them plus one. However, because several processes can read *Number* concurrently, it is possible for several processes to obtain the same number. To break symmetry, we define p_i 's *ticket* to be the pair (*Number*[i], i). Clearly, the tickets held by processes wishing to enter the critical section are unique. We use the lexicographic order on pairs to define an ordering between tickets.

Algorithm 1.1 The Bakery algorithm: code for process p_i , $0 \leq i \leq n - 1$

Initially $Number[i] = 0$ and
 $Choosing[i] = \text{false}$, for i , $0 \leq i \leq n - 1$

(Entry):

- 1 $Choosing[i] := \text{true}$
- 2 $Number[i] := \max(Number[0], \dots, Number[n - 1]) + 1$
- 3 $Choosing[i] := \text{false}$
- 4 **for** $j := 0$ to $n - 1$ ($\neq i$) **do**
- 5 wait until $Choosing[j] = \text{false}$
- 6 wait until $Number[j] = 0$ or $(Number[j], j) > (Number[i], i)$
- 7 **end for**

(Critical Section)

(Exit):

- 8 $Number[i] := 0$

(Remainder)

After choosing its number, p_i waits until its ticket is minimal: For each other process p_j , p_i waits until p_j is not in the middle of choosing its number and then compares their tickets. If p_j 's ticket is smaller, p_i waits until p_j executes the critical section and leaves it. The pseudocode appears in Algorithm 1.1.

We now prove the correctness of the Bakery algorithm. That is, we prove that the algorithm provides the three properties discussed above, mutual exclusion, no deadlock, and no lockout.

Fix an execution α of the algorithm. To show mutual exclusion, we first prove a property concerning the relation between tickets of processes.

Lemma 1.1 In every configuration C of α , if process p_i is in the critical section and for some $k \neq i$, $Number[k] \neq 0$, then $(Number[k], k) > (Number[i], i)$.

Proof Since p_i is in the critical section in configuration C , it finished the for loop, in particular the second wait statement (line 6), for $j = k$. There are two cases according to the two conditions in line 6:

Case 1: p_i read that $Number[k] = 0$. In this case, when p_i finished line 6 (the second wait statement) with $j = k$, p_k either was in the remainder or was not finished choosing its number (since $Number[k] = 0$). But p_i already finished line 5 (the first wait statement) with $j = k$ and observed $Choosing[k] = \text{false}$. Thus p_k was not in the middle of choosing its number. Therefore, p_k started reading the $Number$ array after p_i wrote to $Number[i]$. Thus, in configuration C , $Number[i] < Number[k]$, which implies $(Number[i], i) < (Number[k], k)$.

Case 2: p_i read that $(Number[k], k) > (Number[i], i)$. In this case, the condition will clearly remain valid until p_i exits the critical section or as long as p_k does not choose another number. If p_k chooses a new number, the condition will still be satisfied since the new number will be greater than $Number[i]$ (as in case 1). ■

The above lemma implies that a process that is in the critical section has the smallest ticket among the processes trying to enter the critical section. To apply this lemma, we need to prove that whenever a process is in the critical section its number is nonzero.

Lemma 1.2 If p_i is in the critical section, then $Number[i] > 0$.

Proof First, note that for any process p_i , $Number[i]$ is always nonnegative. This can be easily proved by induction on the number of assignments to $Number$ in the execution. The base case is obvious by the initialization. For the inductive step, each number is assigned either 0 (when exiting the critical section) or a number greater than the maximum current value, which is nonnegative by assumption.

Each process chooses a number before entering the critical section. This number is strictly greater than the maximum current number, which is nonnegative. Therefore, the value chosen is positive. ■

To prove mutual exclusion, note that if two processes, p_i and p_j , are simultaneously in the critical section, then $Number[i] \neq 0$ and $Number[j] \neq 0$, by Lemma 1.2. Lemma 1.1 can then be applied (twice) to derive that $(Number[i], i) < (Number[j], j)$ and $(Number[i], i) > (Number[j], j)$, which is a contradiction. This implies

Theorem 1.1 Algorithm 1.1 provides mutual exclusion.

Finally, we show that each process wishing to enter the critical section eventually succeeds (no lockout). This also implies the no deadlock property.

Theorem 1.2 Algorithm 1.1 provides no lockout.

Proof Consider any admissible execution. Thus no process stays in the critical section forever. Assume, by way of contradiction, that there is a starved process that wishes to enter the critical section but does not succeed. Clearly, all processes wishing to enter the critical section eventually finish choosing a number, because there is no way to be blocked while choosing a number. Let p_i be the process with the smallest $(Number[i], i)$ that is starved.

All processes entering the entry section after p_i has chosen its number will choose greater numbers and therefore will not enter the critical section before p_i . All processes with smaller numbers will eventually enter the critical section (since

by assumption they are not starved) and exit it (since no process stays in the critical section forever). At this point, p_i will pass all the tests in the for loop and enter the critical section, a contradiction. ■

The numbers chosen by the processes can grow without bound during an execution. However, their values are upper bounded by approximately the number of requests to enter the critical section that have occurred so far in the execution.

The Bakery algorithm tolerates failures of the processes under the assumption that a process that crashes returns to its remainder section and eventually all the variables that it writes are reset to zero. Crashed processes that are repaired can start participating in the algorithm again. However, if a process continually fails and restarts, then it could cause other processes to deadlock, as other processes might always read a one from the *Choosing* variable of the faulty process.

Another pleasing feature of the Bakery algorithm is that, unlike some of the previous mutual exclusion algorithms, “turns” are not preassigned to processes. Instead, turns are assigned to processes in the order in which they contend for access to the critical section, and thus contending processes can make progress regardless of the speed of noncontending processes.

1.2.3 Weakening the Shared Variables

The analysis of the Bakery algorithm given above assumes that shared registers are atomic, so that operations appear to occur instantaneously. One pleasant feature of the Bakery algorithm is that such strong memory is not necessary for its correctness. As discussed in more detail in Section 1.3, Lamport introduced weaker forms of memory, regular and safe, which take into account the fact that reads and writes in reality are not instantaneous. Informally, a read of a *regular* register returns either the value of an overlapping write or the value of the latest write that ends before the read begins. A read of a *safe* register that overlaps a write can return any value in the range of the register, but if the read does not overlap a write, then it returns the value of the latest write that ends before the read begins.

As pointed out in [Lamport \[1974a\]](#), nothing in the analysis of the correctness of the Bakery algorithm relies on the shared variables being atomic, and in fact the algorithm is correct even if the shared variables only satisfy the safe property. The *Choosing* variables are binary and thus every read, even if it overlaps a write, gets either the old value or the new value. However, reading a *Number* variable while it is being written could return an arbitrary value, neither the old nor new one.

Consequently, with safe registers, as opposed to atomic or regular, the values of the *Number* variables are not guaranteed to have any relationship to the number

of critical section entry requests that have occurred in the system. [Lamport \[1977\]](#) shows how to keep the *Number* variables incrementing by 1 instead of by arbitrary amounts. Replace each *Number* variable with a sequence of safe variables, each one holding one (nonnegative) digit of the number. To write a value to a *Number* variable, write the digits, in the separate safe variables, from right to left. To read a value from a *Number* variable, read the digits, in the separate safe variables, in the opposite direction, from left to right.

The idea of reading and writing data items in opposite orders is a common theme in [Lamport \[1977\]](#), where it is also applied to solve other distributed synchronization problems, including general readers-writers and producer-consumer. It is also used in an algorithm presented in Section 1.3.

1.3

A Plethora of Registers

[Lamport \[1986d\]](#) delved deeper into a study of the different kinds of registers mentioned in Section 1.2 and explored how stronger ones could be wait-free implemented out of weaker ones (the “base” registers). An implementation is *wait-free* if every execution of an operation on the simulated register completes within a finite number of operations on the base registers, regardless of the behavior of other processes. Wait-free implementations are desirable as they avoid any dependence on solving mutual exclusion or on timing assumptions. In addition to the consistency condition (safe, regular, or atomic), the main parameters of interest are the number of values that can be stored in the register (two or more) and the number of processes that can read the register (one or more). This paper did not consider the possibility of multiple writers accessing the same register.

1.3.1 Increasing the Number of Readers

[Lamport \[1986d\]](#) gave an intuitive algorithm for implementing an m -reader safe register out of a collection of m single-reader safe registers, in which each reader is assigned to one of the registers to read. In order to write the value v to the simulated register, the writer writes v into each of the m single-reader registers, one at a time. In order for the i th reader to read the simulated register, it reads the register to which it is assigned. The simulated register provides the same number of values as do the base registers.

The algorithm is wait-free, as each operation on the simulated register consists of a fixed finite number of operations on the base registers. The safe condition is guaranteed as any simulated read that does not overlap a simulated write returns

the latest value written in its assigned base register, which is the value of the latest simulated write.

As noted in [Lamport \[1986d\]](#), the algorithm also works if the base registers are regular to simulate a regular multi-reader register. The only danger is if the base register read inside a simulated read is concurrent with a base register write inside a simulated write. But the base read is guaranteed to get either the old or new value of the base register, which is also the old or new value of the simulated register.

Unfortunately, if the building block registers are atomic, the algorithm does not ensure that the simulated register is atomic, as it is possible for two consecutive simulated reads that are concurrent with a simulated write to see first the new value and then the old value; this behavior is called a *new-old inversion*. However, subsequent work by [Israeli and Shaham \[1993\]](#) provided a more complicated algorithm to solve this problem. In this algorithm, the readers of the simulated register exchange information among themselves by writing information to shared base registers; this information includes sequence numbers, which grow without bound. A brief description of this algorithm follows.

To avoid new-old inversions, the readers write to each other through additional registers, creating an ordering among them. Before a reader returns from a read operation, it announces the value it has decided to return. A reader reads not only the value written for it by the writer but also the values announced by the other readers. It then chooses the most recent value among the values it has read. In order to decide which value is most recent, every value written is tagged with a sequence number i indicating that the current value is the i th value written by the writer.

With more effort (e.g., [\[Dwork and Waarts 1992, Singh et al. 1994, Dolev and Shavit 1997, Attiya and Welch 1998\]](#)), the sequence numbers can be bounded. However, the requirement for readers to write is provably necessary, as shown next.

Theorem 1.3 In any wait-free implementation of a single-writer multi-reader atomic register from single-writer single-reader atomic registers, at least one reader must write [[Attiya and Welch 2004](#)].

Proof Suppose in contradiction there is such an algorithm for two readers p_1 and p_2 in which no reader writes. Let 0 be the initial value of the simulated register. Since the base registers are single-reader, we can partition them into two sets: S_1 , which are read by p_1 , and S_2 , which are read by p_2 . Consider the execution in which the writer writes 1 to the simulated register. The write algorithm performs a series of writes w_1, w_2, \dots, w_k to the base registers. Let v_j^i be the value that would be returned if p_i were to read the simulated register immediately after w_j , where $i = 1, 2$ and $j = 1, \dots, k$.

For each reader, the atomicity condition requires that there be a point when the writes to the base registers cause the value of the simulated register, as it would be observed by that reader, to “switch” from the old value 0 to the new value 1. Suppose it is after w_a for p_1 for some a ; that is, $v_1^1 = v_2^1 = \dots = v_{a-1}^1 = 0$ while $v_a^1 = v_{a+1}^1 = \dots = v_k^1 = 1$. Similarly, it is after w_b for p_2 for some b . Since the base registers are single-reader, a cannot equal b , as w_a must write to a register in S_1 while w_b must write to a register in S_2 .

Without loss of generality, assume $a < b$. Now suppose that between w_a and w_{a+1} , p_1 reads the simulated register and then p_2 reads the simulated register. The first read will return the new value 1 while the second read will return the old value 0, which is a new-old inversion and violates atomicity, a contradiction. ■

1.3.2 Increasing the Number of Values

When considering safe registers, [Lamport \[1986d\]](#) shows that the standard binary representation of integers suffices for implementing a k -valued register using $\lceil \log_2 k \rceil$ binary registers: In order to write the value v to the simulated register, the writer writes the bits of v 's binary representation into the base registers. In order to read the simulated register, the reader reads all the base registers and returns the represented value. Any number of readers can be supported as long as the base registers support the same number.

Unfortunately, the binary representation algorithm does not work for the regular condition. If a read overlaps multiple writes, it may observe an arbitrary sequence of bits representing a value that is neither the previously written value nor that of any overlapping write.

[Lamport \[1986d\]](#) described an algorithm for implementing a k -valued regular register using k binary regular registers based on the unary encoding of the value. Let b_1, b_2, \dots, b_k be the base registers. To write v , the writer writes 1 into b_v and then writes 0 into b_{v-1} down to b_1 . To read, each reader reads the binary registers in order b_1, b_2, \dots until observing a 1, say, in b_v ; then the reader returns v . Any number of readers can be supported by the simulated register as long as the base registers support the same number of readers.

The correctness of this algorithm is not obvious (and in fact is proved using the formalism discussed in Section 1.4). Part of the analysis shows that the reader is guaranteed to observe a 1 in some base register, and thus the reader will never read a 0 from b_k . Hence b_k is not necessary and the algorithm can be optimized to use only $k - 1$ base registers.

[Lamport \[1986d\]](#) does not consider the analogous problem for atomic registers, other than pointing out that the regular algorithm is subject to new-old inversions

and thus does not work in the atomic case. However, [Vidyasankar \[1988\]](#) gave an algorithm for implementing a k -valued atomic register using binary atomic registers, where $k > 2$. Like the regular algorithm, it is based on the unary representation of the value and thus uses k base registers b_1, b_2, \dots, b_k , but it has a new twist. The value i is represented by a 1 in b_i and 0 in all other base registers. To avoid new-old inversions, two changes are made to the regular algorithm. First, a write operation clears *only* the entries whose indices are smaller than the value it is writing. Second, a read operation does not stop when it finds the first 1 but makes sure there are still zeros in all lower indices. Specifically, the reader scans from the low values toward the high values until it finds the first 1; then it reverses direction and scans back down to the beginning, keeping track of the smallest index observed to contain a 1 during the downward scan. This is the value returned.

1.3.3 Strengthening the Consistency Condition

Implementing a regular binary register out of a safe binary register is easy [[Lamport 1986d](#)]. The only delicate point is that the writer should not write the safe register unless it is actually changing the value. Otherwise, reading and writing the simulated register is done simply by reading and writing the base register.

Using regular base registers to simulate an atomic register is rather involved. [Lamport \[1986d\]](#) presents an algorithm for one reader that implements a k -valued atomic register using two regular registers. One of the regular registers must hold $2k(k + 2)$ values and is written by the writer and read by the reader, while the other one is Boolean and is written by the reader and read by the writer. This algorithm also has a nontrivial correctness proof, which is done using the formalism discussed in Section 1.4.

Note that in the algorithm for implementing an atomic register out of regular registers, the reader and writer communicate with each other. It turns out that it is necessary for the reader to write for such algorithms [[Lamport 1986d](#)].

Theorem 1.4 In any algorithm that implements an atomic register using a finite number of regular registers, one of the regular registers must be written by a reader.

Proof Assume in contradiction that there is such an algorithm in which no reader writes to a base register. First note that without loss of generality, since there is only one writer, we can assume that there is only one base register and the writer writes to it just once. Similarly, we can argue that there is no point in a reader reading the base register more than once.

Suppose the initial value of the simulated register is 0. Consider an execution of the algorithm consisting of the following three consecutive operations on the

simulated register: read 0, write 1, read 1. Each simulated operation consists of a single operation on the base register: first a read, then a write, and then a read. Let v_0 be the state of the base register before the write and v_1 be its state afterward. This execution implies that if the reader reads v_0 (resp., v_1) from the base register, then it must return 0 (resp., 1).

Now consider another execution in which two consecutive reads overlap a write of 1, and in particular, both reads of the base register overlap the write to the base register. Because the base register only satisfies regularity, it is possible for the first read to obtain v_1 and the second read to obtain v_0 . Thus the first read returns 1 and the second read returns 0, which is a new-old inversion and violates atomicity. ■

1.3.4 Increasing the Number of Writers

The case of multiple writers is not considered in [Lamport \[1986d\]](#) and has primarily been studied only for atomic registers, as generalizing the definitions of the safe and regular conditions for multiple writers is more involved. [Vitányi and Awerbuch \[1986\]](#) presented an algorithm for implementing a multi-writer atomic register out of single-writer atomic registers. In this algorithm, the writers communicate among themselves, and thus writers read; part of the information they exchange are sequence numbers that grow without bound.

A simplified version of the algorithm works as follows [Christian Cachin, personal communication, 2006]. Each writer is assigned one base register. All values written are tagged with a sequence number. To write the value v to the simulated register, a writer reads all the registers, chooses the largest sequence number that it observes, increments this sequence number by 1, and writes v together with the sequence number to its register. To read the simulated register, a reader reads all the registers and returns the value associated with the largest sequence number.

With more effort, the sequence numbers can be bounded (e.g., [\[Dwork and Waarts 1992, Dolev and Shavit 1997, Attiya and Welch 1998, Israeli 2005\]](#)). However, the requirement for readers to write is provably necessary, as shown next.

Theorem 1.5 In any wait-free implementation of a multi-writer atomic register from single-writer atomic registers, at least one writer must read.

Proof Suppose in contradiction there is such an algorithm for two writers p_1 and p_2 in which no writer reads. Since the base registers are single-writer, we can partition them into two sets: S_1 , which are written by p_1 , and S_2 , which are written by p_2 . Consider the execution in which p_1 writes 1 to the simulated register, then p_2 writes 2 to the simulated register, and then a reader reads the simulated register; by atomicity the reader must obtain 2.

Since the writers do not read, each one is oblivious to the existence of other writers or readers and thus it always writes the same values to the same base registers, depending solely on its own local history. Also, since the writers write to disjoint sets of base registers, they cannot overwrite each other. As a result, the values of all the base registers in the two executions are the same after the two writes take place. Thus the reader observes the same values in the base registers in the second execution as it does in the first and returns 2, which is a new-old inversion and violates atomicity. ■

1.4

A New Model for Describing Concurrency

Many arguments about the correctness of shared memory algorithms were informal, and Lamport was interested in also having more formal proofs that could be used, for instance, in proving the register algorithms discussed in Section 1.3. Toward this goal, he presented a formalism in Lamport [1986c] for specifying concurrent systems that, unlike previous ones, did not assume that actions are atomic. For concreteness, imagine a system that implements operations on a shared object where the operations consist of lower-level actions and there is a global time model. Since the operations are not atomic, they might overlap in time. One operation execution A can “affect” another operation B if A either precedes or overlaps B . That is, some lower-level action inside A precedes some lower-level action inside B .

More formally, a *system execution* is a triple $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$, where \mathcal{S} is a set of *operation executions* and \rightarrow and \dashrightarrow are precedence relations on \mathcal{S} satisfying

- A1. \rightarrow is an irreflexive partial ordering.
- A2. If $A \rightarrow B$, then $A \dashrightarrow B$ and $B \not\rightarrow B$. (If A precedes B , then A can affect B but B does not precede A .)
- A3. If $A \rightarrow B \dashrightarrow C$ or $A \dashrightarrow B \rightarrow C$, then $A \rightarrow C$. (If A precedes B and B can affect C , then A can affect C , and similarly if A can affect B and B precedes C .)
- A4. If $A \rightarrow B \dashrightarrow C \rightarrow D$, then $A \rightarrow D$. (If A precedes B , B can affect C , and C precedes D , then A precedes D .)
- A5. For any A , the set of all B such that $A \not\rightarrow B$ is finite. (Only a finite number of operation executions precede or overlap A .)

Lamport’s experience is that “proofs based upon these axioms are simpler and more instructive than ones that involve modeling operation executions as sets of

events” [Lamport 1986c]. See below for an example of a correctness proof using this formalism.

This formalism facilitates specifying and reasoning about hierarchical systems. Suppose $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ is a system execution. Let \mathcal{H} be a set whose elements are sets of operation executions from \mathcal{S} ; the elements of \mathcal{H} are called *higher-level operation executions*. An intuitive interpretation is that each higher-level operation in \mathcal{H} is executed via a set of (lower-level) operation executions in \mathcal{S} .

More formally, suppose G and H are in \mathcal{H} . Define $G \xrightarrow{*} H$ to mean that for all $A \in G$ and for all $B \in H$, $A \rightarrow B$; i.e., G precedes H if every lower-level operation in G precedes every lower-level operation in H . Define $G \xrightarrow{*} H$ to mean there exists $A \in G$ and there exists $B \in H$ such that $A \dashrightarrow B$ or $A = B$; i.e., G can affect H if there exists a lower-level operation in A that either can affect or is equal to some lower-level operation in B .

The triple $\langle \mathcal{H}, \xrightarrow{*}, \xrightarrow{*} \rangle$ satisfies Axioms A1–A4 because $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ does. In order to ensure that $\langle \mathcal{H}, \xrightarrow{*}, \xrightarrow{*} \rangle$ is a system execution, it remains to add the following requirement that ensures Axiom A5: Each element of \mathcal{H} is a finite, nonempty subset of \mathcal{S} , and each element of \mathcal{S} belongs to at least one element of \mathcal{H} (but only a finite number). If \mathcal{H} satisfies this condition, then \mathcal{H} is called a *higher-level view* of \mathcal{S} .

A system execution $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ implements a system execution $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xrightarrow{\mathcal{H}} \rangle$ if \mathcal{H} is a higher-level view of \mathcal{S} and for all G and H in \mathcal{H} , if $G \xrightarrow{*} H$, then $G \xrightarrow{\mathcal{H}} H$.

A *system* is defined to be a set of system executions. A system \mathbf{S} implements a system \mathbf{H} if there is a mapping $\iota: \mathbf{S} \mapsto \mathbf{H}$ such that for every system execution $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ in \mathbf{S} , $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ implements $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$.

Lamport [1986d] refines the formalism from [1986c] specifically for analyzing register implementations (cf. Section 1.3). The following additional axioms that must be satisfied by system executions are stated for a single register:

- B0. The set of write operations is $\{V^{[0]}, V^{[1]}, \dots\}$ with $V^{[0]} \rightarrow V^{[1]} \rightarrow \dots$, where $V^{[i]}$ writes the value $v^{[i]}$, and for each read R , $V^{[0]} \rightarrow R$. That is, all the writes are totally ordered, as there is only one writer, and there is an initial write that precedes every read.
- B1. For each read R and write W , either $R \dashrightarrow W$ or $W \dashrightarrow R$. That is, there must be some causal connection between the reads and writes.
- B2. Each read obtains one of the values that may be written to the register.

The axioms for safe, regular, and atomic registers rely on the following definition: A read R is said to see $v^{[i,j]}$, where i is the maximum k such that $R \dashrightarrow V^{[k]}$ and j is

the maximum k such that $V^{[k]} \dashrightarrow R$. That is, R can see traces of the values written by the writes between $V^{[i]}$ and $V^{[j]}$.

- B3. A read that sees $v^{[i,j]}$ obtains the value $v^{[i]}$. That is, a read that does not overlap any writes returns the value of the latest preceding write.
- B4. A read that sees $v^{[i,j]}$ obtains the value $v^{[k]}$, for some k with $i \leq k \leq j$. That is, a read returns the value of an overlapping write or the latest preceding write.
- B5. If a read sees $v^{[i,j]}$, then $i = j$. That is, a read does not (appear to) overlap any write.

A safe register satisfies B0–B3, a regular register satisfies B0–B4, and an atomic register satisfies B0–B5.

Consider the algorithm discussed in Section 1.3.1 for implementing an m -reader safe register out of m single-reader safe registers, denoted v_1, \dots, v_m . Here is an outline of how the formalism is used to prove its correctness. The construction defines a system \mathbf{S} that consists of all system executions $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ such that

- \mathcal{S} consists of reads and writes of the v_i registers.
- Each v_i , $1 \leq i \leq m$, is written by the same writer and is read only by the i th reader.
- For each i and j , $1 \leq i, j \leq m$, if the write $V_i^{[k]}$ occurs, then $V_j^{[k]}$ also occurs and $V_i^{[k-1]} \rightarrow V_j^{[k]}$. That is, a high-level write consists of writing all the v_i 's and one high-level write finishes before the next begins.
- Each v_i is a safe register, i.e., satisfies B0–B3.

The target system \mathbf{H} consists of all system executions $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ that consist of reads and writes to an m -reader safe register. To show that \mathbf{S} implements \mathbf{H} , it must be shown there is a mapping $\iota: \mathbf{S} \mapsto \mathbf{H}$ such that for every system execution $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ in \mathbf{S} , $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ implements some system execution $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$ in \mathcal{H} . Given $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$, the mapping ι is defined as follows. The set of high-level operation executions, denoted $\iota(\mathcal{S})$, is created by having each set of low-level writes $\{V_1^{[k]}, \dots, V_m^{[k]}\}$ form a high-level write $V^{[k]}$ and having each low-level read form a high-level read. Then define $\xrightarrow{\mathcal{H}}$ to be $\xrightarrow{*}$ and $\dashrightarrow^{\mathcal{H}}$ to be \dashrightarrow^* . To finish the proof, it must be shown that $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ is a system execution (satisfies A1–A5), $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ implements $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$, and $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ is in \mathbf{H} (satisfies B0–B3 for the simulated register).

1.5

Sequential Consistency

Lamport coined the term *sequentially consistent* for a multiprocessor that ensures that “the result of any execution is the same as if the operations of all the processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program” [Lamport 1979b]. This correctness condition had been considered in previous work (e.g., [Dijkstra 1971, Owicki and Gries 1976, Lamport 1977]) but not under this name. As we explain below, sequentially consistent memory is weaker than atomic memory but is incomparable with regular and safe memory.

Even though a multiprocessor contains sequential processes, the sequential nature of the processes alone is not sufficient to ensure sequential consistency for the multiprocessor, since memory operations are not instantaneous. However, in Lamport [1979b], two necessary and sufficient conditions are given to ensure that the multiprocessor is sequentially consistent.

The first condition, R1, is that each process must issue its memory fetch and store requests in the order of its program. This is fairly intuitive, but does rule out some optimizations that are benign in the single-process situation.

The second condition, R2, is that all memory requests to the same memory module (or cell) must be serviced in the order in which the requests are made. In other words, there is a single FIFO queue for each memory module into which requests are put. Here’s an example execution showing that without R2, sequential consistency is violated. Suppose there are two shared variables, a and b , both initially 0, and two processes, p_1 and p_2 . Let p_1 ’s sequential program consist of writing 1 to a and then reading b , while p_2 ’s sequential program consists of writing 1 to b and then reading a . Consider the execution in which p_1 sends a request to write 1 to a to memory module 1, which is currently busy and thus puts the request in a queue, and then p_1 sends a request to read b to memory module 2, which immediately services the request. Then suppose p_2 sends a request to write 1 to b to memory module 2, which immediately services the request, and then p_2 sends a request to read a to memory module 1. If memory module 1 is still busy, then p_2 ’s request is put in a queue, which is not the one containing p_1 ’s request. Suppose memory module 1 eventually services p_2 ’s read request before p_1 ’s write request (see Figure 1.1). Then p_2 ’s read of a gets the old value 0 instead of the new value 1. There is no sequentially consistent way to reconcile this behavior with the fact that p_1 ’s read of b gets the old value 0.

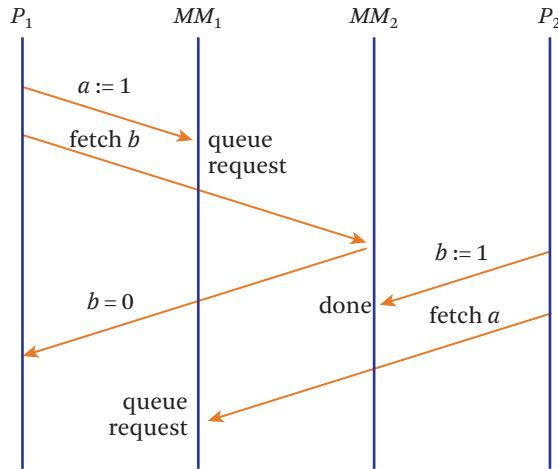
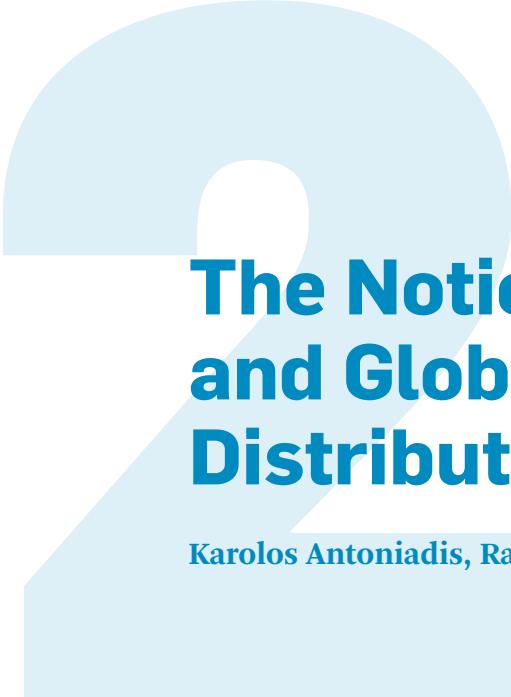


Figure 1.1 Diagram to justify the necessity of condition R2.

To prove the sufficiency of R1 and R2, first define a relation \rightarrow on memory requests that orders requests by the same process in the order in which the process issues them and orders requests by different processes on the same memory module in the order in which the requests are enqueued. It is straightforward to see that \rightarrow is a partial ordering. Then it can be shown that in any execution of the multiprocessor, each fetch and store operation has the same effect as if the operations were executed instantaneously in any total order that is consistent with the \rightarrow partial order.

Tying back to the registers discussed in Section 1.3, sequential consistency is weaker than atomicity. Both conditions require a sequential ordering of the operations, but for atomic registers the ordering must reflect that of all non-overlapping operations, while for sequential consistency it is only for operations by the same process. Thus an execution in which one process writes a new value to the register and then later a second process reads the old value of the register may be sequentially consistent but not atomic. On the other hand, sequential consistency is incomparable with both regularity and safety, as the previous execution is not safe (and thus not regular), but the following execution is regular (and thus safe) but not sequentially consistent: the reader does two consecutive reads, both of which overlap with a write, where the first read returns the new value and the second read returns the old value.

A tremendous amount of work has been done on sequential consistency. A few (nonexhaustive) directions of interest include lower bounds on the latency of operations on sequentially consistent memory (e.g., [Lipton and Sandberg \[1988\]](#)); performance differences between sequential consistency and other conditions (e.g., [Attiya and Welch \[1994\]](#)); identifying programming patterns that provide the illusion of sequential consistency on top of weaker conditions (e.g., [Adve and Hill \[1990\]](#), [Gibbons et al. \[1991\]](#)); determining whether a multiprocess actually provides sequential consistency (e.g., [Gibbons \[1997\]](#)); and understanding the complexity of deciding whether a protocol provides sequential consistency (e.g., [Condon and Hu \[2003\]](#)).



The Notions of Time and Global State in a Distributed System

Karolos Antoniadis, Rachid Guerraoui

2.1

Introduction

Our day-to-day life is filled with a smorgasbord of *events*: a child drops a ball, a phone rings, etc. Ordering such events by global time is simple. For example, if the phone rings at 2:00 PM and the child drops the ball at 5:00 PM, then we know that the phone rang *before* the child dropped the ball. But what if events occur very close to each other in time? It is still easy to order events if we are present when they occur. For instance, we can easily recognize that we received an email before our colleague coughed. Things get tricky if we want to order events occurring close to each other in different parts of the world. To give an example, Figure 2.1 depicts two persons, John and Alice, experiencing events. In this example, John buys a chocolate bar at time 10:59 PM based on his watch, while Alice gets the newspaper at time 11:00 PM based on her watch. But if Alice's watch is 3 minutes ahead then in reality, the purchase of the newspaper occurred before the purchase of the chocolate.

Nevertheless and as we will see later on, if the two persons interact in some way, then we can potentially infer the order of some events. For instance, if Alice throws a banana peel and then John passes by and slips on it, then we can infer that the throwing of the banana peel *happened before* the slip. In other words, we cannot really deduce whether the purchase of the chocolate or the newspaper took place first, but we can be certain that Alice first threw the banana peel and thereafter John slipped on it. We depict this "happens before" relation with an arrow between the

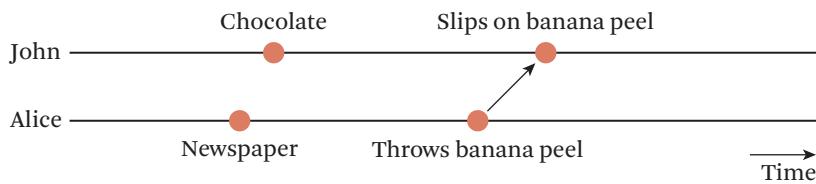


Figure 2.1 Two persons, John and Alice, experiencing events.

related events in Figure 2.1. Notice that in Figure 2.1, as well as all other figures in this chapter, time is depicted from left to right.

Similar to real life with different people, ordering events in distributed computing systems is a difficult issue. Distributed computing systems consist of a set of independent Turing machines (also called *processes* or *nodes*) that communicate with each other. Processes perform *events* that in a distributed system correspond to performing a computation, sending or delivering messages. We could try to introduce physical clocks to the processes to order events, but it would be difficult to keep them synchronized in order to extract meaningful information from them. It is natural for clocks to drift apart, so some clocks might move faster than others.

Lamport devised an approach to order events without resorting to physical time. His approach captures what he called *logical time*. Roughly speaking, logical time orders events based on *causality*: if some event possibly causes another event, then the first event *happens before* the other. We describe logical time in Section 2.2.

Lamport introduced his notion of logical time in 1978 in his celebrated paper “Time, Clocks, and the Ordering of Events in a Distributed System” [Lamport 1978b]. This paper is mostly known for defining logical time, as well as the concept of causality. However, of equal if not greater importance was the introduction (in the same paper) of a way to implement an arbitrary *state machine* in a distributed setting. In other words, he devised an approach that is based on a logical ordering of events and that can be used to implement in a distributed fashion every possible algorithm. We describe the distributed state machine abstraction idea and an algorithm to implement any algorithm in a distributed setting in Section 2.3.

Lastly, in Section 2.4 we describe the very concept of global state and how to retrieve it in a distributed system. This was first introduced in the influential paper by Chandy and Lamport [1985] titled “Distributed Snapshots: Determining Global States of Distributed Systems.” This paper utilized ideas from logical time to capture the concept of *global state* in a distributed system.

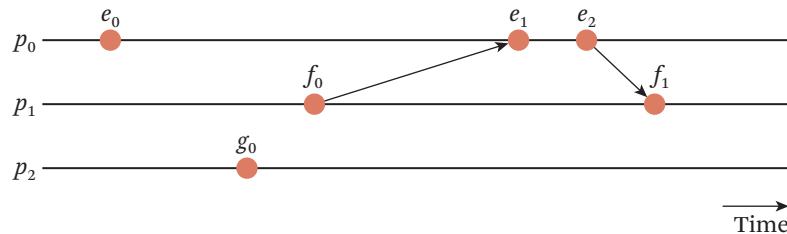


Figure 2.2 Three processes and their respective events.

2.2

The Notion of Logical Time

A distributed system consists of a set of independent processes (Turing machines) that communicate with each other, typically by exchanging messages¹ as depicted in Figure 2.2. Three types of events can take place in a distributed system from the perspective of every process: (i) perform some local computation, (ii) send a message, and (iii) deliver a message.

In Figure 2.2, we depict three processes p_0 , p_1 , and p_2 together with their respective events. Process p_0 performs some local computation (event e_0), then delivers a message (event e_1) and afterward sends a message (event e_2). Similarly, process p_1 sends a message (event f_0) and delivers a message (event f_1). Process p_2 performs a single computation (event g_0).

As we pointed out before, it is challenging to order events based on the actual physical time when the events occur. Sometimes, we are not even interested in which event took place first. For example, in Figure 2.2, whether e_0 took place before or after g_0 is perhaps unlikely to be of any consequence. Furthermore, even if we could augment each process with a physical clock, physical clocks could drift apart from each other, making it challenging to order events. Lamport realized that we can order events in a different but still useful way: based on *causality*. This way, if an event e possibly causes event f , then we can order these two events e and f and argue that e happened before f . For instance, if an event corresponds to a delivery of a message, this means that the event that sent this message preceded the delivery event. Knowledge of the order of events can be utilized to build distributed state machines. Knowing the order of events can also be useful in debugging distributed systems, garbage collecting old versions of data, etc. Lamport devised an algorithm

1. A multiprocessor can also be considered a distributed system. However, processes in a multiprocessor communicate by reading/writing to shared memory (see Chapter 1).

that captures what is called *logical time* and can express the possible causal order between events.

Remarks. Lamport's main inspiration for logical time was a report titled "The Maintenance of Duplicate Databases" by [Johnson and Thomas \[1975\]](#). Johnson and Thomas introduced in this report the notion of timestamps in order to keep different copies of a database eventually consistent.² They used timestamps to order the operations issued to a distributed database. However, the timestamps they proposed were associated with clocks of processes and hence could get out of synchrony, thus violating causality. Furthermore, their work was specific to distributed databases. In contrast, logical time is applicable to every distributed system. As a side note, Johnson and Thomas pointed out in the same work that when a network partition occurs, either consistency is violated or we cannot provide availability. Eric Brewer made a similar observation 25 years later that is today known as the CAP theorem [[Brewer 2000](#), [Gilbert and Lynch 2002](#)].

Another source of inspiration for logical time was Lamport's knowledge of special relativity. According to special relativity there is no fixed order on the exact time an event took place: different observers could experience the same event at different times. Lamport realized that the same principle applies to the events of a distributed system; however, we can still argue whether one event could have potentially caused another one.

2.2.1 Causality and Logical Time

Lamport's idea behind logical time is both simple and clever. Lamport realized that certain events in a distributed system are associated with each other through a "happens before" relation, while others are not. We cannot relate all events with regard to "happens before" since some events are concurrent (e.g., events e_0 and g_0 of Figure 2.2 are concurrent). Nevertheless, there are cases in which we can clearly state that one event happened before another event. For example, all the events that are occurring at a single process are ordered sequentially (e.g., a process first performed a computation, then the process sent a message, etc.). In other words, the "happens before" relation induces a partial order.

Given two events e, f , $e \rightarrow f$ denotes that an event e precedes another event f and we say that e "happens before" f . The \rightarrow relation captures *causality* since it

2. *Eventually consistent* means that if processes stop performing updates to the database for enough time, then eventually all the copies of the database contain the exact same data.

means that e potentially caused f . We can infer whether an event e precedes an event f based on the following rules:

- If one process performs an event e before starting an event f , we can infer that e precedes f ($e \rightarrow f$).
- If e corresponds to the sending of a message from some process and f corresponds to the respective delivery of this message by some other process, then we can establish that $e \rightarrow f$.
- Naturally, transitivity applies and therefore if an event g exists such that $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

Logical time refers to the fact that we can utilize the \rightarrow relation to order events without knowing the actual physical time of the events.

2.2.2 An Algorithm to Capture Causality

A process in a distributed algorithm could utilize the notion of logical time to wait until something happens, before it proceeds to perform a computation. Therefore, capturing the aforementioned notion of causality algorithmically can prove useful. (We will see an example of the usefulness of capturing causality in the next section.)

To capture causality, Lamport devised a simple algorithm (see Algorithm 2.1). This algorithm can be used to augment other distributed algorithms and allow them to know if an event e precedes another event f . The algorithm operates as follows. Each process p is associated with a *logical clock* t_p that “ticks” at every event. In other words, each event is associated with a number (also known as *Lamport timestamp*) that is given by the logical clock. Each time a process p performs an event, p increments its clock t_p . Furthermore, each message is augmented with this clock value, so when a process p sends a message, p also includes the value of t_p . On the other end, when a process p receives a message, p sets its timestamp to the maximum between its local clock timestamp t_p and the received timestamp and then p increments t_p by one. Assuming the timestamp of each event is given by a function $C : \text{events} \rightarrow \mathbb{N}$, Algorithm 2.1 guarantees that for every two events e and f , if $e \rightarrow f$, then $C(e) < C(f)$. Figure 2.3 depicts the execution of Algorithm 2.1 and the timestamp assigned to each event.

Total Order. We can extend the partial order of events to a total order by extending the timestamp with a unique identifier. This identifier could be, for example, the process identifier. For instance, an event in such a setting would be (e, id_e) where e is an event and id_e is an identifier. Then for any two events (e, id_e) and (f, id_f) we can say that $(e, id_e) < (f, id_f)$ if $e \rightarrow f$, or if $e \not\rightarrow f$ and $f \not\rightarrow e$ and $id_e < id_f$.

Algorithm 2.1 Lamport's timestamps (for a process p)

```

procedure ONCOMPUTATION()
   $t_p \leftarrow t_p + 1$ 
end procedure

procedure ONSEND (msg,  $p_k$ )            $\triangleright$  send message msg to process  $p_k$ 
   $t_p \leftarrow t_p + 1$ 
  send([msg,  $t_p$ ],  $p_k$ )
end procedure

procedure ONRECEIVE()                   $\triangleright$  on receiving some message
   $[m, t] \leftarrow receive()$ 
   $t_p \leftarrow \max(t, t_p) + 1$ 
end procedure

```

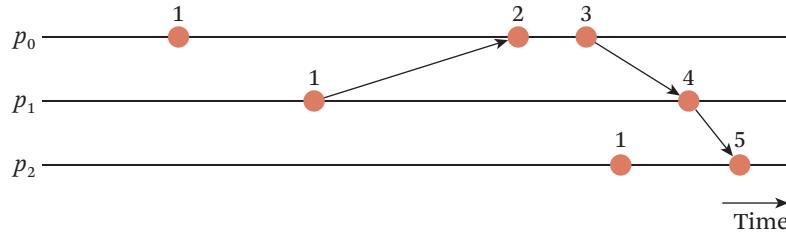


Figure 2.3 The timestamps assigned to events by using Algorithm 2.1.

By extending the partial order to a total order, we can utilize the total order to implement an arbitrary distributed state machine, as we show in the next section.

2.2.3 Impact of Logical Time

Lamport's work on logical time had an enormous impact in both theory research and practical systems.

An important extension emerging from Lamport's clocks is the notion of *vector clocks*. Vector clocks can capture the *lack* of causality while Lamport's clocks cannot. In other words, if $e \rightarrow f$, then $C(e) < C(f)$, but $C(e) < C(f)$ does not mean that $e \rightarrow f$. For example, in Figure 2.3, the first event of process p_2 has a smaller timestamp than the third event of process p_0 ; however, this does not mean that p_2 's first event occurred before the third event of p_0 . Vector clocks were devel-

oped independently in the late 1980s by [Fidge](#) [1988], [Mattern](#) [1989], and [Schmuck](#) [1988].

In practical systems, ideas based on logical time have been extensively used for replicating data (e.g., [ISIS](#) [Birman 1986, Birman et al. 1991]), debugging purposes (e.g., [ShiViz](#) [Beschastnikh et al. 2016]), garbage collection [[Liskov and Ladin](#) 1986, [Terry et al.](#) 1995], and reducing message communication (e.g., [Bayou](#) [[Terry et al.](#) 1995]). Among others, *version vectors* [[Parker et al.](#) 1983] that utilize logical time have been used in key-value stores, such as [Dynamo](#) [[DeCandia et al.](#) 2007], [Riak](#) [2019], and [Voldemort](#) [2019], as well as file systems (e.g., [Coda](#) [[Satyanarayanan et al.](#) 1990]) to handle write conflicts. Recently, causal consistency has gained attention. Causal consistency [[Ahamad et al.](#) 1995] is a consistency model inspired by causality and is the strongest consistency model that can tolerate network partitions [[Attiya et al.](#) 2015]. A multitude of recent distributed storage systems [[Didona](#) 2018, [Du](#) 2014, [Lloyd et al.](#) 2011, [Lloyd et al.](#) 2013, [Zawirski et al.](#) 2015] provide causal consistency.

2.3

The Distributed State Machine Abstraction

As with logical time, Lamport's inspiration behind the distributed machine abstraction was the Johnson and Thomas report [[Johnson and Thomas](#) 1975]. Johnson and Thomas tried to keep different copies (i.e., replicas) of a database eventually consistent, while allowing each replica to independently introduce updates. Lamport came up with a clean abstraction to solve this problem: the *distributed state machine abstraction*. Lamport's main insight behind the distributed state machine abstraction is that, by applying commands in the same order at all the replicas of a distributed system, we can obtain a universal approach for keeping the replicas consistent with each other.

Roughly speaking, Lamport's idea states that all the processes together simulate a state machine. This can be achieved if each process applies the exact same commands of the machine and in the same order as every other process. Specifically, this can be done with the following approach. Each process contains a local copy of the state machine. All processes initialize their state machine to the same initial state. Processes apply commands to their state machine such that each process applies the exact same commands in the same order.

The main difficulty of such an approach is ordering the commands, and this is where logical time comes into play. Although the approach seems simple, Lamport was the first one to conceptualize it, and it had an immense impact in practical systems.

We call an algorithm that implements a universal state machine in a distributed system by utilizing replication a *state machine replication* (SMR) algorithm. SMR algorithms are useful in practice in order to implement fault-tolerant systems. Implementing an SMR algorithm in a system where failures can occur (e.g., crashes) is a difficult problem [Lamport 1998a, Liskov and Cowling 2012, Ongaro and Ousterhout 2014].

To demonstrate the universality of the SMR approach, Lamport first presented a simple SMR algorithm that does not tolerate process failures. Equipped with the notions of logical time and SMR, the problem of keeping the replicas of a database consistent with each other is simply a matter of tracking the logical time of the updates that are separately generated and received by them. At the time there was no solution for keeping the copies of a database consistent.

Additionally, Lamport's SMR algorithm assumes that the communication links between processes are *perfect* and *FIFO* [Cachin et al. 2011]. By *perfect* we mean that if a process p sends a message m to a process q , then q eventually delivers m ; and by *FIFO* we mean that if a process p sends a message m_1 before sending message m_2 to some process q , then q cannot deliver m_2 before having delivered m_1 .

We continue by presenting Lamport's SMR algorithm that uses logical time. We conclude this section by discussing the impact Lamport's idea had in the theory of distributed computing, as well as in practical systems.

2.3.1 SMR Algorithm

We consider a state machine sm that serves commands c_1, \dots, c_k , and we want to have a distributed algorithm in which each process simulates the execution of this state machine. In the following description we assume an *asynchronous* system with n processes p_0, p_1, \dots, p_{n-1} . Each process has a state machine sm in an initial state. Additionally, each process contains a *log* of unexecuted (i.e., not yet applied) commands. Figure 2.4 depicts the data structures each process maintains locally, where the right box corresponds to the log of not yet applied commands (i.e., commands c_3 and c_{13} have not yet been applied to sm). Each entry in the log contains a Lamport timestamp, and the log is kept sorted according to timestamp order.

Finally, each process contains an array of n timestamps, the latest timestamp received from each other process. Process p_i maintains its own logical clock in $timestamps[i]$. Timestamps are maintained based on the Lamport timestamp algorithm presented in Section 2.2.2. In what follows, when we say that a message containing a timestamp t is being sent, this means that t corresponds to the timestamp given by the logical clock of the process for the “sending of the message” event.

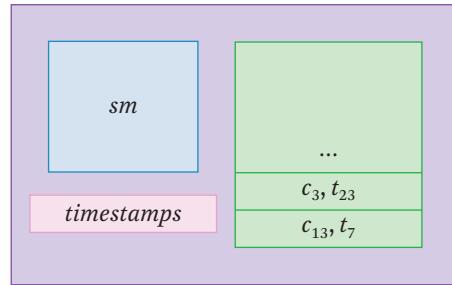


Figure 2.4 The data structures that a process maintains locally.

The main idea behind the SMR algorithm is that a process p can apply a command c by first broadcasting a message m that contains command c to all the other processes. Process p can apply command c to p 's state machine when p receives from all the other processes at least one message carrying a higher timestamp than m .

Specifically, the SMR algorithm operates using the following simple rules:

- When a process p_i receives a message containing a timestamp t from another process p_j , p_i updates the $timestamps[j]$ element of the array to contain t (recall that links are FIFO).
- To apply a command c_k , process p_i sends message $[c_k, t]$ to all the other processes where t is the timestamp of the message. Then, process p_i creates an entry $[c_k, t]$ and appends this entry to the log of not yet applied commands.
- When a process p_i receives a message $[c_k, t]$, process p_i appends the $[c_k, t]$ entry in its log of not yet applied commands and subsequently sends an acknowledgment message (i.e., $[ack, t']$) to every other process (i.e., broadcasts the message). Naturally, it is the case that $t' > t$. By sending an acknowledgment message, a process informs other processes that it has received the command.
- A process p_i applies a command c_k if the first (i.e., oldest) entry in p_i 's log of not yet applied commands is $[c_k, t]$. However, p_i can only apply command c_k if p_i has received messages from every other process with timestamps greater than t (i.e., all the timestamps in the $timestamps$ array contain values $> t$). If this is the case, p_i can apply c_k on its local state machine and remove the entry from its log.

Algorithm 2.2 presents the algorithm of process p_i .

Algorithm 2.2 An SMR algorithm using logical time

```

1   $\triangleright$  local variables
2   $sm_i \leftarrow init$   $\triangleright$  initial state machine
3   $log_i \leftarrow \emptyset$   $\triangleright$  log of not yet applied commands
4   $timestamps[n] \leftarrow \{0, \dots, 0\}$ 

5  procedure APPLYCOMMAND( $c_k$ )
6     $timestamps[i] \leftarrow timestamps[i] + 1$ 
7     $\forall p_j \neq p_i, send([c_k, timestamps[i]], p_j)$ 
8     $log_i.append([c_k, timestamps[i]])$ 
9  end procedure

10 procedure ONRECEIVE()
11    $\triangleright p_i$  on receiving a message from process  $p_j$ 
12    $msg \leftarrow receive()$ 
13   if  $msg = [c_l, t_j]$  is a command message then
14      $log_i.append([c_l, t_j])$ 
15      $timestamps[i] \leftarrow \max(timestamps[i], t_j) + 1$ 
16      $timestamps[j] \leftarrow t_j$ 
17      $\forall p_j \neq p_i, send([ack, timestamps[i]], p_j)$ 
18   else  $\triangleright msg = [ack, t_j]$  is an acknowledgment (ack) message
19      $timestamps[j] \leftarrow t_j$ 
20   endif
21 end procedure

22 procedure CONSTANTCHECK()
23   while true do
24      $[first.command, first.ts] \leftarrow log_i.peek()$ 
25      $can.apply \leftarrow true$ 
26     for  $j = 0; j < n; j = j + 1$  do
27       if  $j \neq i$  and  $first.ts \geq timestamps[j]$  then
28          $can.apply \leftarrow false$ 
29         break
30       endif
31     end for
32     if  $can.apply$  then
33        $sm.apply(first.command)$ 
34        $log_i.remove([first.command, first.ts])$ 
35     endif
36   end while
37 end procedure

```

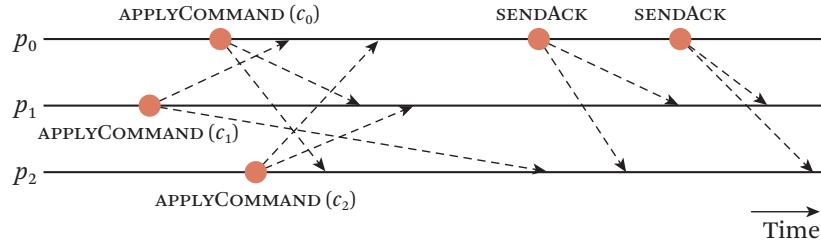


Figure 2.5 Execution of the SMR algorithm.

Procedure `CONSTANTCHECK` of Algorithm 2.2 runs in the background and checks (line 27) if there are entries from all processes in the log. In this case, the head of the log (i.e., the entry whose timestamp is the lowest) is applied.

The algorithm applies the exact same commands and in the same order at each process, since a process p applies a command c only if p has received messages with higher timestamps from all other processes. This guarantees that no command preceding c in the timestamp order exists, since p 's communication with all other processes preserves FIFO order.

Figure 2.5 depicts an execution of Algorithm 2.2 in which three processes request to apply different commands concurrently. Although requests are issued concurrently, the processes apply the commands in the same order. Recall that logs are kept ordered by the global timestamp order. Assume that the timestamp ordering of the three commands in this scenario is c_0, c_1, c_2 . For a process p to apply a command that is in its log, p needs to receive messages from all the other processes with greater timestamps. For instance, when process p_1 calls `APPLYCOMMAND(c_1)`, p_1 just knows of its own command c_1 . In order for p_1 to apply c_1 , p_1 would need to receive messages with greater timestamps from processes p_0 and p_2 . Since the communications links are FIFO, process p_0 would first send its command c_0 to p_1 and afterward the acknowledgment message to p_1 . Therefore, p_1 would realize that it first needs to apply command c_0 of process p_0 before applying command c_1 .

Remarks. The knowledgeable reader might think that if we assume a setting without failures, implementing an SMR algorithm can be easily done by using a leader-based approach. With a leader-based approach we can trivially achieve total ordering of the commands. For example, if we assume that a single process of the system acts as a *leader*, then it is easy to devise an SMR algorithm. Assuming that process p_0 is the leader, such an algorithm could operate as follows. Every process gets

informed on what commands to apply and in which order by process p_0 . Specifically, before a process applies a command, the process first sends the command to the leader p_0 . Process p_0 can then order the commands it has received and dictate the execution order of these commands. Subsequently, p_0 can inform all the other processes about this execution order. Since there are no failures, we are guaranteed that p_0 is always up and running. However, the problem of implementing an SMR algorithm remains challenging if we exclude the trivial leader-based solution. Nevertheless, by utilizing logical time we can implement an SMR algorithm.

The reason Lamport proposed a non-leader-based SMR algorithm probably has to do with the fact that he was still thinking about the problem in the context of databases where we want to keep replicas consistent. Nevertheless, note that the SMR algorithm described in this section is quite effective as the basis for a fault-tolerant solution when clocks are synchronized. Lamport's first way of tackling fault tolerance was in fact to assume synchronized clocks (and separately to work on clock-synchronization protocols, which are mentioned in the next chapter).

2.3.2 Impact of the Distributed State Machine Abstraction

Although in retrospect the distributed state machine abstraction is conceptually simple, it had a tremendous impact on both theory and practice.

The distributed state machine abstraction appeared repeatedly in Lamport's subsequent work. After the "Time, Clocks, and the Ordering of Events in a Distributed System" [Lamport 1978b], Lamport proceeded to work on various SMR implementations. The culmination of Lamport's work led to SMR algorithms for both benign and Byzantine settings. See Chapters 3 and 4 for some of these algorithms.

In practice and with the surge of cloud computing, the SMR abstraction has been extensively used in practical systems. Major cloud infrastructure companies report using SMR in their data centers for lock services, data replication, and atomicity management. Likewise, major storage vendors are using it to manage data redundancy.

2.4

The Notion of Distributed Global State

A *distributed snapshot* refers to taking a "photograph" of a distributed system. This snapshot comprises the state of each process at the moment the snapshot is taken. The state of a process corresponds to the state of the local data structures of the process and what the process is executing (i.e., program counter). In other words, a snapshot captures the *global state* of the system. In this section, we describe an algorithm that is able to capture a distributed snapshot and discuss how it relates to

the notion of the distributed global state. Before we start, we present the following analogy to better convey the meaning of a snapshot.

A parent had three children: Bob, Dan, and Tom. The parent wanted to instill financial awareness to his three sons and decided to give them an allowance of \$20 each. Immediately after, the children started exchanging money with each other for toys, etc. One day, the father wanted to check on his children to see how much money each of them had. During that day, the father asked Bob, then Dan, and at the end Tom how much money they currently had. At the end of the day and based on the responses of the children, the father deduced that Bob, Dan, and Tom have \$10, \$30, and \$50, respectively. Surprisingly, the children had \$30 more than what they received from their father! The father quickly realized that the children did not magically become richer. Instead, what happened was that after asking Dan how much money he had and before asking Tom, Dan gave his \$30 to Tom. Ideally, the father would want to get an instantaneous view of his sons' money boxes in which the total money is \$60. In other words, the father wants to get a snapshot of his sons' money boxes.

In a distributed setting, the children are the processes and the father is an algorithm that takes a snapshot in order to perform some action. For example, instead of checking how much money each child had, we would like to find out the CPU utilization of each process when the snapshot is taken. However, compared with the aforementioned analogy, taking a snapshot in a distributed system is a more difficult problem. For instance, we cannot gather all the processes together in the same room and “ask” them what their current state is, something that the father could easily do. Such an approach would be analogous to having a synchronous system in which all processes provide their state at a specific point in time.

Here we consider an asynchronous system. In an asynchronous system, the notion of global state is elusive. For instance, it is challenging to define what a global state is, since we have no notion of global time. Ideally, we want to take a snapshot of a system while the system is executing a distributed algorithm, but without impeding the progress of the algorithm. In such a setting, it is hard to conceive what the global state of a system is, since the system is constantly in fluctuation (i.e., processes keep performing events).

[Chandy and Lamport \[1985\]](#) devised an algorithm to capture a snapshot in an asynchronous distributed system in their influential “Distributed Snapshots: Determining Global States of Distributed Systems.” In that work, they managed to capture the notion of a distributed global state. We explain a simplified version of their algorithm in the next section. In Section [2.4.2](#) we describe the impact of their work.

Algorithm 2.3 Snapshot algorithm (for process p_i)

```

1   $\triangleright$  local variables
2   $stateRetrieved_i \leftarrow false$ 
3  procedure STARTSNAPSHOT()
4     $state_i \leftarrow p_i.getState()$ 
5     $stateRetrieved_i \leftarrow true$ 
6     $\forall j \neq i, send(marker, p_j)$ 
7  end procedure

8  procedure ONRECEIVEMARKER()
9    if  $stateRetrieved_i \neq true$  then
10       $state_i \leftarrow p_i.getState()$ 
11       $stateRetrieved_i \leftarrow true$ 
12       $\forall j \neq i, send(marker, p_j)$ 
13    end if
14  end procedure

```

2.4.1 The Distributed Snapshot Algorithm

Similarly to Section 2.3, we consider an asynchronous system with no failures and with reliable and FIFO communication links. Furthermore, we assume that each process provides a *getState* function that can take a local snapshot of the local state (i.e., local data structures, program counter, etc.) of the process instantaneously. We present the algorithm for taking a distributed snapshot in Algorithm 2.3.

The algorithm works as follows: a *single* central process³ p decides to take the snapshot and starts by executing STARTSNAPSHOT. Process p takes a snapshot of its local state (line 4), then sets the *stateRetrieved* variable to *true* (line 5) so that p does not retrieve its local state again at some later point in time (line 9), and then p sends a message that contains a *marker* (line 6) to every other process. The *marker* message is simply used to inform other processes to take a snapshot. When a process receives a *marker* message, it executes the ONRECEIVEMARKER function that retrieves the local state if it has not already retrieved it (line 9) and sends a *marker* message to all the other processes.

We remark that Algorithm 2.3 can augment a distributed algorithm in order to capture the global state of the system during the algorithm's execution without impeding the algorithm's progress (i.e., the distributed algorithm can keep

3. The central process can be assigned before the distributed system starts. Another way to decide on a central process can be by using the SMR algorithm of Section 2.3.

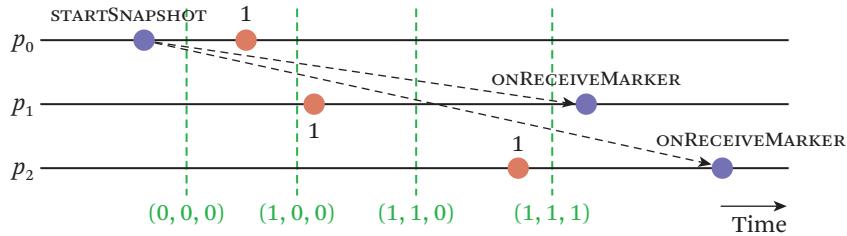


Figure 2.6 The distributed snapshot algorithm does not necessarily capture a global system state that actually occurred during an execution.

performing events while the snapshot is being taken). Additionally, note that Algorithm 2.3 describes when each process should take a snapshot of its local state but not how all the local states are assembled together to form the global state. This is beyond the scope of this section, but could be easily accomplished by having each process broadcast its local state after retrieving it.

Assume a distributed algorithm executing in a system with n processes that uses logical clocks to timestamp its events. Then we can think of a distributed snapshot taken by Algorithm 2.3 as a vector $(ts_0, ts_1, \dots, ts_{n-1})$ of n elements (also known as a *cut*) where the local state of a process p_i is retrieved after an event e timestamped with ts_i and before the event immediately following e in process p_i . Figure 2.6 depicts the execution of an algorithm (red events at each process). Before process p_0 performs an event, p_0 starts capturing a snapshot of the system. The *marker* messages sent by p_0 are delivered to processes p_1 and p_2 after each of them has performed one event. The resulting distributed snapshot corresponds to the cut $(0, 1, 1)$. However, such a global state never existed in the system. The green-dashed vertical lines in Figure 2.6 show the global states that existed in the system. These global states correspond to the cuts $(0, 0, 0)$, $(1, 0, 0)$, $(1, 1, 0)$, and $(1, 1, 1)$. Surprisingly, the snapshot algorithm captures a global state that did not exist. It is challenging to capture a global state at a specific instant while processes keep performing events.

However, the snapshot algorithm satisfies two useful properties. The first property has to do with all the events that take place between the moment process p_0 invoked `STARTSNAPSHOT` and the moment all the processes have retrieved their local state (i.e., have called the `getState` function). We can reorder these events and execute the distributed algorithm based on this new order of events such that the snapshot returned by the snapshot algorithm corresponds to a global state during the distributed algorithm's execution. For instance, in Figure 2.6 if the events of processes p_1 and p_2 were ordered before the event of process p_0 , then the cut $(0, 1, 1)$

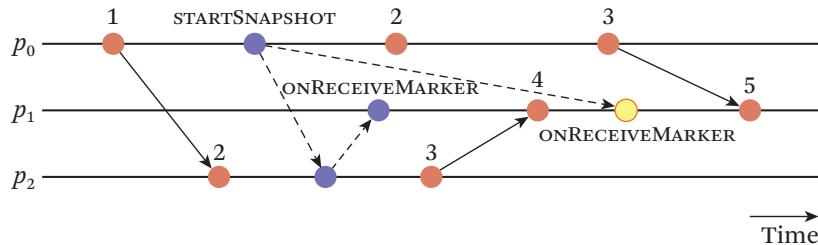


Figure 2.7 Taking a distributed snapshot during the execution of an algorithm.

would correspond to a global state during the execution of the system. The second property states that if the global state captured by the snapshot algorithm contains an event e in the local state of some process, then all the events that causally precede e are captured in the local states of some other processes that belong to the snapshot. In other words, the notion of a distributed global state is intertwined with causality.

In Figure 2.7 we present another example that depicts the execution of an algorithm while a distributed snapshot is being taken (blue and yellow events). In this example, p_0 is the process that initiates the snapshot by invoking `STARTSNAPSHOT`. By doing so, a *marker* message is sent to all the other processes. Process p_2 is the first process that receives the *marker*, so p_2 takes a snapshot of its local state and sends a *marker* to the other processes. Notice that since the communication links are FIFO the *marker* sent by p_2 reaches p_1 before the message that p_2 sent during the event with timestamp 3. When p_1 receives the *marker*, p_1 takes a snapshot of its local state and sends *marker* messages as well. For clarity and since each process has taken a snapshot of its local state, we omit the additional *marker* messages that are sent by the processes. Afterward, when p_1 receives the *marker* from p_0 (the yellow event), process p_1 does not perform anything since p_1 has already taken a snapshot of its local state.

Finally, in Figure 2.8 we present the same execution as in Figure 2.7 and show possible snapshots that can and cannot be returned by the algorithm. The red curved dashed lines correspond to snapshots that cannot be returned by the snapshot algorithm. For example, the snapshot that corresponds to the cut $(2, 4, 2)$ cannot be returned by the algorithm, since p_1 includes the state of the process after the event with timestamp 4 has occurred but does not include the event with timestamp 3 of process p_2 that causally precedes p_1 's event. Similarly, a snapshot that corresponds to the cut $(2, 5, 3)$ cannot be returned by the algorithm since the

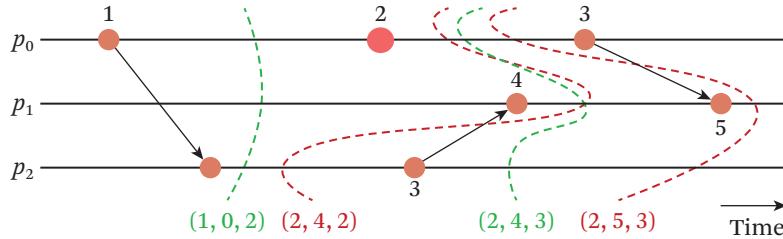


Figure 2.8 Valid (green) and invalid (red) snapshots based on causality. Algorithm 2.3 returns only valid snapshots.

cut does not capture causality. The green curved dashed lines correspond to cuts that can be returned by the snapshot algorithm, such as $(1, 0, 2)$ and $(2, 4, 3)$.

Remarks. Someone might think that we could take a distributed snapshot by utilizing an SMR algorithm. An idea would be for processes to just apply commands to the global (i.e., simulated) state machine. This way, if a process wants to get the global state of the system, the process just needs to retrieve the state of the global state machine. Such an approach is unsatisfactory since it does not really capture the state of the process, but instead captures the state of the global state machine. Additionally, such an approach is less practical since each possible event has to go through the SMR algorithm.

2.4.2 Impact of Distributed Global State

The distributed snapshot algorithm might take a snapshot of a global state of the system that never existed in reality. This raises the question whether such an algorithm is even useful.

However, as we mentioned before, the snapshot algorithm satisfies two useful properties. One of them states that there exists a reordering of the events that take place during the retrieval of the snapshot such that the returned global state exists in an execution corresponding to these reordered events. We present an example of this property in Figure 2.9, assuming that the snapshot algorithm returned a global state s'_i . Specifically, Figure 2.9(i) depicts the sequence of global states during the execution of the distributed algorithm. Note that state s'_i does not exist in the sequence of global states in Figure 2.9(i). Furthermore, global state s_{start} corresponds to the state where the snapshot algorithm is first invoked by process p_0 , while s_{end} corresponds to the state where all processes have retrieved their local state. If we reorder the events that took place during the execution of the snapshot

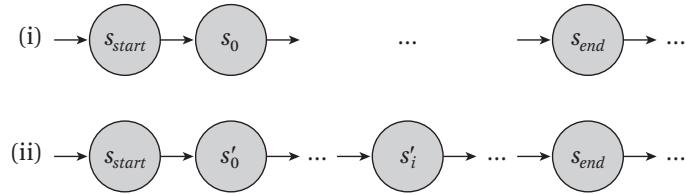


Figure 2.9 Sequence of global states (ii) corresponds to the states generated by an execution of the reordered events of execution (i).

algorithm, we get a possible different sequence of states (e.g., s_0 is different from s'_0) that is depicted in Figure 2.9(ii) and in which state s'_i exists. The snapshot algorithm guarantees that immediately after the reordering of the events the system is in the same state as before the reordering; hence both sequences of states end with the state s_{end} . We refer the interested reader to the original paper [Chandy and Lamport 1985] for a formal proof for why the events-reordering property holds.

We can now see how the aforementioned property of the snapshot algorithm is useful in practice. The snapshot algorithm's property is useful in identifying whether a *stable property* of a system holds. A stable property is a property that if it holds in some state of the system, it holds in every subsequent state of the system. For instance, whether a deadlock has occurred is a stable property since if a deadlock occurs, the system remains forever deadlocked. We can use the distributed snapshot algorithm to detect a stable property by retrieving the global state and examining whether the stable property holds in the retrieved global state. If the stable property holds in the retrieved global state, then this means that the stable property holds in the system. For instance, if the stable property holds in the retrieved state s'_i in Figure 2.9(ii), this means that the stable property also holds in s_{end} and any subsequent state of s_{end} and hence the stable property holds in the system. To give an example, if we retrieve the global state and we detect a deadlock in this state, we can infer that the system is deadlocked. Detecting stable properties during the execution of an algorithm is extremely useful. Among others, such a detection mechanism can be used for checkpointing and failure recovery [Koo and Toueg 1987, Prakash and Singhal 1996].

To conclude, the notion of a global state in distributed computing had a huge impact in the last three decades, ranging from distributed [Mattern 1989] to concurrent [Afek et al. 1993, Jayanti 2005] systems.

2.5

Conclusion

After getting acquainted with the notions of time and global state in a distributed system, they become clear and might appear obvious in retrospect. This does not mean this was indeed the case. The work of Lamport successfully democratized these notions. Lamport's paper "Time, Clocks, and the Ordering of Events in a Distributed System" had tremendous impact on introducing the notion of time in a distributed system. Most importantly, the distributed state machine abstraction approach that stemmed from this paper spawned new research areas and has been widely implemented in various settings and deployed in a plethora of practical systems. Finally, the "Distributed Snapshots: Determining Global States of Distributed Systems" paper by Chandy and Lamport introduced the notion of a global state in a distributed system.

3 Byzantine Faults

Christian Cachin

3.1

Introduction

This chapter covers Lamport's most prominent works addressing attackers and malicious acts in distributed computing systems. One would think that dealing with adversaries lies in the domain of computer security and cryptography—both areas outside Lamport's core domain of *concurrency*, according to this book's title. But Lamport has initiated the study of distributed protocols in such adversarial settings through the formulation of the *Byzantine generals problem* and thereby founded a fruitful research program that has been vibrantly active for several decades. It is less widely known that he also made seminal contributions to cryptography and computer security in the early days of these fields.

In more detail, Lamport's work on the *Byzantine generals problem* [Lamport et al. 1982] has influenced generations of researchers and practitioners and represents one of his most important contributions. The intuitive story of the paper around the problem of reaching agreement, with all the connotations of the term *Byzantine*, has attracted and fascinated many people. However, the problem of reaching agreement in the presence of faults had already been the subject of two of his earlier papers that grew out of the NASA-sponsored Software Implemented Fault Tolerance (SIFT) project, whose goal was to build a resilient aircraft control system that tolerated faults of its components. Lamport's first research paper on SIFT appeared in 1978, with Wensley et al. [1978], and the second one in 1980, with Pease and Shostak [Pease et al. 1980]. But widespread interest in the problem arose only after the publication of the "The Byzantine Generals Problem" in 1982. Indeed, Lamport's intention behind writing this paper was to popularize the agreement problem. To use his own words [Lamport 2019]:

I have long felt that, because it was posed as a cute problem about philosophers seated around a table, Dijkstra's dining philosopher's problem received

much more attention than it deserves. (For example, it has probably received more attention in the theory community than the readers/writers problem, which illustrates the same principles and has much more practical importance.) I believed that the problem introduced in [Lamport et al. 1982] was very important and deserved the attention of computer scientists. The popularity of the dining philosophers problem taught me that the best way to attract attention to a problem is to present it in terms of a story.

In order to reduce the problem to its most extreme case, the story of the Byzantine generals lets the faulty generals behave maliciously and adversarially. Lamport and coauthors originally did not think it was realistic to assume an attacker had actually gained access and was trying to create disruption from within, when their protocols would run in an industrial control system or even on an aircraft. This was rather chosen as a realistic and intuitive abstraction for covering all incorrect behavior that a protocol participant might exhibit. Much later only has the term *Byzantine fault tolerance* gained its current meaning of dealing with actually malicious parties and tolerating their actions.

Still, Lamport was very much aware of the need to secure distributed computer systems from attackers. In concurrent work addressing problems in security and cryptology, he introduced *one-time signatures* [Lamport 1979a], which are today known as *Lamport signatures* in cryptography, and a novel protocol for *password authentication using one-way functions* [Lamport 1981], which pioneered many others.

In the following, we highlight three prominent works of Lamport that deal with adversarial protocol participants: Byzantine agreement, clock synchronization in the presence of arbitrary faults, and digital signature schemes built from one-way functions.

3.2

Byzantine Agreement

The two papers by Lamport, Shostak, and Pease [Pease et al. 1980, Lamport et al. 1982] on Byzantine agreement address the same problem: Given n processes p_1, \dots, p_n with one input value each, how to synchronize them with a protocol that only uses point-to-point messages such that all processes agree on an output vector V of n values that correctly represent the inputs. During the protocol execution, an unknown set of t processes is *faulty* or *Byzantine* and the remaining $n - t$ processes are *correct*. The Byzantine processes may behave in arbitrary and adversarial ways, as if intending to prevent the correct processes from reaching agreement.

3.2.1 Definitions

The Byzantine agreement problem can be described in three slightly different ways, which are all equivalent to each other in the *synchronous* model considered by Lamport and coauthors. There are small and conceptually simple transformations to convert one variant to another. The three forms are called *interactive consistency*, the *Byzantine generals problem*, and *Byzantine agreement*. (In *asynchronous* systems, however, where no common clock exists, these three notions differ considerably; failure to distinguish these nuances has led to some confusion in the literature. We consider only synchronous systems in this chapter.)

Interactive Consistency. The agreement problem described in the paragraph above is called *interactive consistency*: Each process inputs a value, say, p_i inputs v_i , and each process outputs a vector V of such values. It satisfies these properties:

Agreement. Each correct process outputs the same vector of values.

Validity. If a correct process outputs V and a process p_i is correct and inputs v_i , then $V[i] = v_i$.

In the SIFT project building an aircraft controller, interactive consistency was intended to synchronize the different processes, where each process would supply its locally read sensor values as input. After reaching agreement on an output vector like this, each process would then derive its actions deterministically from the output. This ensures that the actions of all processes remain synchronized and safe.

Byzantine Generals Problem. The Byzantine generals problem [Lamport et al. 1982] is best understood as a *broadcast primitive*, where a designated *sender* process p_s starts with an input value v and all other processes have no input; at the end all processes output some value. The goal is that the sender conveys its input to all processes in a *reliable* way such that all processes output the same value. For this reason we call it *Byzantine broadcast* here, also because it is reminiscent of other notions for (Byzantine) reliable broadcast [Cachin et al. 2011].

In the Byzantine generals formulation, there is one *commander* acting as the sender p_s and all other processes are *lieutenants* that should obey the order of the commander. More abstractly, a protocol for the Byzantine generals problem or for Byzantine broadcast ensures

Agreement. Each correct process outputs the same value.

Validity. If the sender process, p_s , is correct and has input v , then every correct process also outputs v .

This notion was formalized by [Lamport et al. \[1982\]](#) and used to describe the impossibility results and several protocols. In the introduction of the paper, however, the story describes that “the multiple divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general” and that “they must decide upon a common plan of action.” This is again the notion of interactive consistency.

The relation between the two primitives is straightforward, which is why Lamport et al. introduce the simplification. Namely, given a solution for Byzantine broadcast, interactive consistency can be achieved by running n separate instances of Byzantine broadcast in parallel, one for each process, and that process acts as the sender in its instance. In the other direction, the relation is already evident from the definition, as interactive consistency is the composition of n Byzantine broadcasts, one for each process as a sender. [Lamport et al. \[1982\]](#) explain all of this in the classic paper.

Byzantine Agreement. The notion of *Byzantine agreement* or *Byzantine consensus* stands between interactive consistency and the Byzantine generals problem. In this primitive, *every* process inputs a value and *every* process outputs again a *single* value. Formally, it satisfies

Agreement. Each correct process outputs the same value.

Validity. If all correct processes input the same value v , then every correct process outputs v .

Note that *validity* in Byzantine agreement may not be useful for practical applications since it says nothing about the situation where the correct processes start with diverging inputs. If not all processes supply the same input, then Byzantine agreement may choose a value that comes out of thin air; for example, it could be some default value \perp to indicate that no agreement was reached or it could be something made up by the faulty processes. On the other hand, if all correct processes input the same value, they appear to “agree” already, so why should they run an agreement protocol? The answer is, clearly, that running the protocol helps them because they don’t *know* that they agree a priori.

The validity notion of Byzantine agreement has many subtle aspects. One could simplify it by additionally requiring that the output value was input by a *correct* process. However, then the set from which those values are chosen starts to play a role [[Fitzi and Garay 2003](#)] and impacts the resilience of possible protocols (i.e., the relation between the number of faulty processes, t , and n). In other contexts, it has been suggested to *externalize* the decision on which agreement values are “valid”

and permit any value to be output, as long as it satisfies some predicate known to each process [Ben-Or and El-Yaniv 2003, Cachin et al. 2001].

Despite these and other issues with validity in Byzantine agreement, this formulation of the agreement problem has become the preferred one of the three variants. This may also be because the equivalent notion of *consensus* in asynchronous systems, even if processes are only subject to crashes, plays a fundamental role in distributed computing [Herlihy and Wing 1990, Chandra and Toueg 1996].

To implement Byzantine agreement from interactive consistency, the processes first run interactive consistency to agree on an input value for each process. Then every process runs a deterministic procedure locally that selects the output according to the validity notion required by Byzantine agreement. Implementing Byzantine agreement from a primitive for Byzantine broadcast (i.e., a protocol for the Byzantine generals problem) works similarly.

To realize Byzantine broadcast for sender p_s from Byzantine agreement, process p_s first sends its value directly to each process in the system. Then all processes run Byzantine agreement once, whereby a correct process inputs the value received from p_s . This protocol can be extended to an implementation of interactive consistency as explained under the description of the Byzantine generals problem.

3.2.2 Implementations

Protocols for Byzantine agreement start from a model that makes authenticated point-to-point links available, such as provided by physical channels linking all pairs of processes or by using an open communication network and protecting the communication using a cryptographic message-authentication code (MAC) that prevents the Byzantine processes from tampering with messages among correct processes.

The necessary and sufficient conditions on t and n that make Byzantine agreement possible differ based on whether one considers *oral messages* or permits the use of *digital signatures*. With oral messages, the integrity of a statement made by one process and forwarded by a Byzantine process cannot be guaranteed, in the sense that the recipient process cannot determine whether the forwarding process has altered the statement. Digital signatures, on the other hand, provide data authentication based on a cryptographic public-key/private-key pair for each process.

Digital Signatures. A *digital signature scheme* provides two operations, $sign_i$ and $verify_i$. Every invocation of $sign_i(v)$ specifies the index i of a process, takes a bit string $v \in \{0, 1\}^*$ as input, and returns a signature $\sigma \in \{0, 1\}^*$ with the response. Only p_i may invoke $sign_i$. The operation $verify_i$ takes a bit string σ that should be verified

and a bit string v as parameters. It returns a Boolean value with the response. Implementations of these methods satisfy that $\text{verify}_i(\sigma, v)$ returns TRUE for any $i \in \{1, \dots, n\}$ and $v \in \{0, 1\}^*$ if and only if p_i has executed $\text{sign}_i(v)$ and obtained σ before; otherwise, $\text{verify}_i(\sigma, v)$ returns FALSE. Every process may invoke verify . Therefore, it is not possible for any Byzantine process to cook up a statement and a valid signature on it from a correct process.

Impossibility Results. The most fundamental impossibility result concerning Byzantine agreement states that Byzantine broadcast with oral messages (in the synchronous model) requires $n > 3t$. In other words, no protocol exists for $n \leq 3t$. To see why, consider the Byzantine broadcast problem for the special case $n = 3$ and $t = 1$ as follows [Lamport et al. 1982].

The commander C should send either *attack* or *retreat* to the two lieutenants, L_1 and L_2 . The argument introduces two scenarios that are *indistinguishable* from each other by one of the correct participants (L_1) because it receives exactly the same information. In the first scenario, the sender C is correct and inputs *attack*. Then L_1 receives *attack* directly from C but L_2 is Byzantine and changes all protocol messages to L_1 as if C had input *retreat*. By the validity property, L_1 must output *attack*. However, in a second scenario, C is Byzantine and acts toward L_1 as if its input were *attack* and toward L_2 as if its input were *retreat*. Since L_2 is correct, it behaves toward L_1 as if C had input *retreat*. The correct L_1 observes the same information as in the first scenario and decides *attack*. A dual argument for the second scenario then implies also that L_2 decides *retreat*, since C acts toward L_2 with input *retreat*. Then the two correct lieutenants output different values, which violates the agreement condition and shows that no such protocol exists.

The result can be extended to arbitrary $n \leq 3t$ by considering groups of processes [Lamport et al. 1982]. Many subsequent results have used this proof method to establish impossibilities and lower bounds on the resilience of distributed protocols for other tasks and under different assumptions.

Intuitively, process L_1 cannot obtain the correct output in the example because there are only two other processes that interact with L_1 , each one suggesting a different binary value. With one more process, L_1 could decide for the value that it learns from a majority and break the tie. The underlying problem is the same as in an error-correcting code for tolerating garbled symbols, specifically, in a repetition code that requires two correctly transmitted symbols for each corrupted symbol. The relation between agreement problems and error-correcting codes was explored in depth later, for example, by Friedman et al. [2007].

Table 3.1 Bounds on the resilience of (synchronous, deterministic) Byzantine agreement problems

Problem	Signed Messages	Oral Messages
Interactive consistency	$n > 2t$	$n > 3t$
Byzantine broadcast (Byzantine generals)	$n > t$	$n > 3t$
Byzantine agreement	$n > 2t$	$n > 3t$

The lower bound of $n > 3t$ extends in a straightforward way to the problems of interactive consistency and Byzantine agreement. Note that the argument exploits the mutability of oral messages. In fact, if digital signatures are available, then the Byzantine broadcast problem can be implemented for any $n > t$. Protocols for interactive consistency and for Byzantine agreement with signatures require $n > 2t$, however.

The difference arises, intuitively, because there is much less ambiguity about the output value in a Byzantine broadcast if signatures are available than about the decision in an agreement problem. Since the sender p_s may digitally sign its input, every process can recognize this value as correct from the signature. A typical protocol would spread around the signed input until every correct process is guaranteed to have it [Lamport et al. 1982, Sec. 4]. This mechanism also prevents disagreement among the receivers because if p_s were to sign multiple inputs, then the dissemination protocol ensures that every correct process would get all of them. The processes would then decide that the sender was faulty and output a default value.

In Byzantine broadcast without signatures, on the other hand, and in the other two variants (interactive consistency and Byzantine agreement), no such knowledge from a single sender exists. In these forms of the problem, *every* process has an input value or is free to echo what it heard directly from p_s in Byzantine broadcast. Hence digital signatures cannot “isolate” the correct output value in a Byzantine broadcast, and the protocols must resort to decisions among a correct majority.

Table 3.1 summarizes these bounds on the resilience of Byzantine agreement variants, with digitally signed messages and oral messages (and deterministic protocols in the synchronous model).

Protocols. The most intuitive protocol for one of the Byzantine agreement problems in the two classic papers solves Byzantine broadcast. It uses digital signatures, takes $t + 1$ rounds, and incurs exponential communication cost in n .

Algorithm 3.1 Byzantine broadcast using digital signatures for p_i

```

1   $V \leftarrow \emptyset$ 
2  on input  $v$  do ▷ Sender (commander)  $p_s$  only
3     $\sigma_s \leftarrow \text{sign}_s(\text{VALUE} \parallel v)$ 
4    send round-0 message  $[v, 0; p_s; \sigma_s]$  to  $p_1, \dots, p_n$ 
5  end on
6  for  $k \leftarrow 1, \dots, t + 1$  do
7    on receiving a round- $(k - 1)$  message  $M$  from  $p_j$  do
8       $[v, m; p_s, p_{i_1}, \dots, p_{i_m}; \sigma_s, \sigma_{i_1}, \dots, \sigma_{i_m}] \leftarrow M$ 
9      if  $m = k - 1 \wedge i_m = j \wedge \text{validate}(M) \wedge v \notin V$  then
10         $V \leftarrow V \cup \{v\}$  ▷ Collect all values in  $V$ 
11         $\sigma_i \leftarrow \text{sign}_i(M)$ 
12         $M' \leftarrow [v, m + 1; p_s, p_{i_1}, \dots, p_{i_m}, p_i; \sigma_s, \sigma_{i_1}, \dots, \sigma_{i_m}, \sigma_i]$ 
13        send round- $k$  message  $M'$  to  $p_\ell \in \{p_1, \dots, p_n\} \setminus \{p_i, p_{i_1}, \dots, p_{i_m}\}$ 
14      endif
15    end on
16  end for
17  if  $V = \{w\}$  then
18    output  $w$ 
19  else ▷  $|V| > 1$ 
20    output  $\perp$ 
21  endif

```

Recall that in a Byzantine broadcast (or the Byzantine generals problem), a designated sender p_s (the commander) conveys an order to the other processes (the lieutenants). Any t of the n processes may be Byzantine. The protocol introduced in the paper proceeds in $t + 1$ synchronized rounds. Basically, the sender signs the value it wants to broadcast and sends it to all processes. Whenever a process receives a properly signed message, it remembers the value and signs the message again. Then the process appends its own signature to the message and forwards it to those processes that have not yet signed it. At the end, after $t + 1$ rounds, a process examines its set of received values: if it contains exactly one entry, the process outputs this value as its decision, otherwise, it chooses a default value. The pseudocode is shown in Algorithm 3.1. As an optimization, in the last round ($k = t + 1$) a process can skip lines 11–13.

The function $\text{validate}(M)$ is defined on a message M as follows:

```

function validate( $M$ )
   $[v, m; p_s, p_{i_1}, \dots, p_{i_m}; \sigma_s, \sigma_{i_1}, \dots, \sigma_{i_m}] \leftarrow M$ 
  if  $m = 0$  then
    return  $verify_s(\sigma_s, \text{VALUE} \| v)$ 
  else
     $M' \leftarrow [v, m - 1; p_s, p_{i_1}, \dots, p_{i_{m-1}}; \sigma_s, \sigma_{i_1}, \dots, \sigma_{i_{m-1}}]$ 
    return  $verify_{i_m}(\sigma_{i_m}, M') \wedge validate(M')$ 
  end if
end function

```

To see why this algorithm is correct, let us first examine validity: If the sender p_s is correct and has input v , then no protocol message with a value different from v will be accepted by *validate*. Since p_s also sends v in its first protocol message to all processes, it follows that $V = \{v\}$ for each correct process when the protocol terminates. Hence every correct process outputs v . The *agreement* property requires that no two correct processes output different values. Since the function determining the output from variable V is deterministic, it is sufficient to show that all correct processes have the same V after round $t + 1$. Suppose some correct p_i adds v_i to its V in some round. Then we must show that another correct process p_j also adds v_i to its set V . There are two cases to consider: (1) If p_i receives v_i from p_1 in round 1, then it sends a round-1 message containing v_i to p_j and p_j adds v_i after receiving this. (2) If p_i receives v_i in round $k \geq 2$, then p_i has received a round- $(k - 1)$ message with k signatures. If p_j is one of the signers, then it already has added v_i to its set V according to the protocol. Otherwise, we consider again two cases: (2a) If $k \leq t$, then p_i forwards v_i to p_j in its round- k message; p_j receives this in round $k + 1$ and adds v_i to its V . Else (2b) we have $k = t + 1$ and among the $t + 1$ processes that issued the $t + 1$ signatures in the round- t message received by p_i , there must be at least one other correct process. This process has also forwarded v_i to p_j in a valid message and the set V at p_j therefore contains v_i . Hence the two processes output the same value.

This protocol incurs exponentially large communication cost and time complexity in n (for $t = O(n)$) because, in the worst case, each process p_i may receive a round- k message signed by a set of $k + 1$ processes for each one of the $(k + 1)!$ possible paths across the process set that such a message from the sender may have taken until it reached p_i . In fact, all Byzantine agreement protocols in the early papers of Lamport, Shostak, and Pease [Pease et al. 1980, Lamport et al. 1982] used

this protocol structure, which has also been called *exponential information gathering* [Lynch 1996].

In the model with oral messages, a variant of this protocol is described by Lamport et al. [1982], also implementing Byzantine broadcast. Informally, the modification introduces a step of choosing a value among multiple ones that a process receives from others. This provides an example of how a *majority decision* ensures safety in an agreement algorithm.

Soon after the pioneering papers appeared, Byzantine agreement protocols with polynomial cost were developed for the models with signatures and without, first by Dolev and Strong [1982, 1983]. The same authors also showed formally that $t + 1$ rounds are required for deterministic synchronous Byzantine agreement [Dolev and Strong 1983].

Many more protocols for Byzantine agreement were developed later; their number is so large that one cannot even start with most important ones here. The longest-lasting open problem posed directly by Lamport in the two early papers was closed only about 15 years later, when Garay and Moses [1998] described the first efficient (polynomial-time) protocol with oral messages that achieves optimal resilience $n > 3t$ and an optimal latency of $t + 1$ rounds.

The following section further examines distributed agreement protocols in asynchronous and partially synchronous networks. These algorithms use slightly different approaches. Protocols that completely disregard synchrony also introduce randomization as a fundamental mechanism to relax the deterministic models discussed here.

3.3 Byzantine Clock Synchronization

In order to achieve the synchrony necessary to operate a distributed system, early works by Lamport focused on the problem of synchronizing the clocks of a group of processes. In a classic paper Lamport and Melliar-Smith [1985] formalized this problem for the first time and initiated a long series of works that extended and refined their ideas.

Their model assumes that each process has an associated clock; this clock can be read by all processes in the system as in a shared-memory multiprocessor computer. This assumption stands in contrast to models used later, where each process has exclusive access to its own clock. However, this variation does not lead to a fundamental change in the ideas used by the respective clock synchronization

protocols. Faulty clocks and processes may be Byzantine and subject to arbitrary faults.

To formalize the problem, Lamport and Melliar-Smith assume that all clocks are initially synchronized and that the clocks of correct processes run at approximately the same rate, that is, their differences are bounded by some small value δ per unit of real time. The second assumption they introduce concerns the message-transmission times: the processes must know how fast they can convey information to each other. In particular, a correct process can obtain the difference between its own clock and that of another correct process within a time interval bounded by ϵ .

Synchronizing the clocks means (1) that the clock values at the correct processes are approximately the same at each instant, and (2) that the clock at every correct process differs at most by a bounded value from real time. In their pioneering paper, Lamport and Melliar-Smith introduce two fundamental methods for addressing clock synchronization.

The first algorithm is based on *averaging* the time values read from other processes. Very informally speaking, this means that each process periodically reports its clock reading to all others; each process then sets its own clock to the average of the reported clock values. Unreasonably large differences between a reported value and the clock of the receiver process are discarded. This can be done safely, assuming that bounds are placed on the message transmission times and on the difference between the clocks among the correct processes. The protocol itself uses a simple structure with direct messages among the processes, only conveying clock readings to each other.

The second algorithm introduces the idea of setting the clock of a process to the *median* among reported values; this reduces the influence of outliers compared to the average and ensures that, with n processes and at most $t < n/3$ faulty ones, the derived clock value always lies between the real clock value of two correct processes. Processes use in this solution a subprotocol for interactive consistency, as in the earlier papers on agreement, in order to exchange clock values. The subprotocol is extended by measures that take into account the time spent executing the protocols.

3.4

Digital Signatures

Around the same time as he was working on distributed agreement and replication, Lamport developed an interest in the nascent field of *public-key cryptography*. Modern cryptography was born in 1976 with the paper “New Directions in

Cryptography" by [Diffie and Hellman \[1976\]](#), which introduced the field of public-key cryptography with the novel concepts of trapdoor one-way functions, public-key cryptosystems, and digital signatures. Although the paper postulated that all these primitives could be realized from computationally hard problems that arise in number theory and discrete mathematics, it did not provide concrete realizations except for the interactive key-agreement protocol that now bears the name of the paper's authors.

Digital signatures were envisaged by Diffie and Hellman to have applications to authenticating digital documents, securing messages sent over a network from tampering, and protecting remote logins to multiuser computers. (All these predictions, and many more, have come true.) And while the key-agreement protocol of Diffie and Hellman had been closely related to public-key cryptosystems, and therefore a candidate implementation seemed feasible, there was no similar construction for a digital signature anywhere on the horizon.

However, Lamport had developed such a primitive from more traditional cryptographic building blocks soon after hearing about the invention of public-key cryptography. The scheme is actually contained in the paper of Diffie and Hellman and is attributed there to Lamport [[Diffie and Hellman 1976](#), Sec. IV], but he actually never published it himself. Lamport devised a related signature scheme later and described it in a technical report [[Lamport 1979a](#)].

Cryptographic Definition of Digital Signatures. Recall the notion of a *digital signature scheme* from Section 3.2.2, with two operations, *sign* and *verify*. This is an idealized formulation (in the sense of [Dolev and Yao \[1983\]](#)) made for studying protocols because it omits the key material, but not suitable for describing and analyzing actual constructions. Here we redefine a signature scheme, as in modern cryptography, to consist of a triple $(keygen, sign, verify)$ of efficient algorithms. The *key generation* algorithm *keygen* outputs a public-key/private-key pair (pk, sk) . The signing algorithm *sign* takes as input the private key and a message $m \in \{0, 1\}^*$, and produces a signature $\sigma \in \{0, 1\}^*$. The verification algorithm *verify* takes the public key, a message m , and a putative signature σ , and outputs a Boolean value (TRUE or FALSE) that indicates whether it accepts or rejects the signature. The signature is *valid* for the message when *verify* accepts. All signatures produced by the signing algorithm must be valid.

A digital signature scheme is secure against *existential forgery* if no efficient adversary \mathcal{A} can output any message together with a valid signature that was not produced by the legitimate signer. More formally, \mathcal{A} is given pk and is allowed to request and obtain signatures on a sequence of messages of its choice, where any

message may depend on previously observed signatures. If \mathcal{A} can output a valid signature on a message whose signature it never obtained, then the adversary has successfully *forged* a signature. A signature scheme is *secure* if any efficient \mathcal{A} can forge a signature only with negligible probability. (Formally, *efficient* means with running time polynomial in a security parameter κ of the scheme, and *negligible* is smaller than any polynomial fraction of κ .)

Lamport Signatures. Lamport's signature scheme from 1976 is based on a generic cryptographic *one-way function*. This is a function $F : \{0, 1\}^k \rightarrow \{0, 1\}^k$ that is efficiently computable but hard to invert on average. More precisely, this means for $x \in_R \{0, 1\}^k$, i.e., a random, uniformly chosen k -bit string, and $y = F(x)$, and any efficient adversary \mathcal{A} that is given y , the probability that $\mathcal{A}(y)$ outputs x is negligible. In practice, a one-way function is often implemented by a collision-free hash function, such as SHA-2 or SHA-3, or constructed from a block cipher like AES.

Key generation for a Lamport signature on ℓ -bit messages first selects 2ℓ random bit strings of length k each, which together form the private key

$$sk = (x_1[0], x_1[1], \dots, x_\ell[0], x_\ell[1]).$$

Then the one-way function F is applied to each $x_j[b]$ to compute

$$y_j[b] = F(x_j[b]), \quad \text{for } j = 1, \dots, \ell \text{ and } b \in \{0, 1\}.$$

The public key is the list of 2ℓ such bit strings of length k each,

$$pk = (y_1[0], y_1[1], \dots, y_\ell[0], y_\ell[1]).$$

To sign an ℓ -bit message $m = (m_1, \dots, m_\ell)$, algorithm $sign(sk, m)$ returns

$$\sigma = (x_1[m_1], \dots, x_\ell[m_\ell]),$$

i.e., one preimage under F as prepared during key generation for each message position, selected by the bit value in that position.

For validating a signature $\sigma = (\sigma_1, \dots, \sigma_\ell)$ on m , algorithm $verify(pk, m, \sigma)$ applies F itself to the bit strings in the signature and checks if all of them match pk , as determined by the bits in the message, according to

$$y_j[m_j] \stackrel{?}{=} F(\sigma_j), \quad \text{for } j = 1, \dots, \ell.$$

Discussion. The signature scheme exploits a simple but powerful idea, which is to reveal secrets selectively determined by a protocol input. The fixed length is not a drawback. In practice, messages of arbitrary length are first compressed with

a collision-free cryptographic hash function to the ℓ -bit inputs of the signature scheme.

Still, the public key and the secret key each have size $2\ell k$, which is not practical. [Merkle \[1980\]](#) reduced the public-key size to one k -bit string by constructing a tree, with all 2ℓ strings in the private key at the leaves. Each node in this tree, which now bears Merkle's name, is the output of a cryptographic hash function applied to its children. During a *sign* operation, extra values in the tree are added and allow *verify* to establish the same integrity properties as in Lamport's scheme. In particular, for authenticating a leaf node, Merkle's scheme considers the path from a leaf node to the root and adds the values of all sibling nodes along the path to the signature.

In contrast to the general notion of a public-key digital signature, however, Lamport's scheme can be used *only once* for a given key pair. Namely, if any index $j^* \in \{1, \dots, \ell\}$ would be reused by different *sign* operations for messages m' and m'' that differ in position j^* and also elsewhere, then the adversary could exploit this and forge a valid signature on a message m^* that has never been signed. Such an m^* can be obtained, for instance, by flipping bit j^* in m' .

For this reason, Lamport signatures are also called *one-time signatures*. Even if a larger key is prepared to sign multiple messages, this will be exhausted in linear time, proportional to the number of signing operations. If multiple uses are foreseen, then the signing operation must keep state—a further drawback that is problematic in practice compared to generic public-key signatures.

To see why the scheme satisfies the definition of a cryptographic signature scheme, observe first that for every signature produced by *sign*, algorithm *verify* returns TRUE. For the unforgeability property, assume that no index in the key is used twice. Toward a contradiction, suppose adversary \mathcal{A} has produced a forgery in terms of a message m^* and a signature σ^* such that $\text{verify}(pk, m^*, \sigma^*) = \text{TRUE}$. Since no index in σ^* has been used before, \mathcal{A} has successfully inverted F on the bit strings in pk . However, this is not possible if F is one-way.

The RSA cryptosystem [[Rivest et al. 1978](#)] provided the first practical realization of digital signatures that fully matched the concept proposed by Diffie and Hellman. Many other constructions based on trapdoor one-way functions built from number-theoretic primitives followed. Lamport's one-time signatures, on the other hand, were not developed much further due to their impracticality with large keys and the state to be kept by the signer.

Revival. Around 1990, surprising results by [Naor and Yung \[1989\]](#) and [Rompel \[1990\]](#) showed that, in principle, one-way functions were sufficient to build cryptographically secure digital signatures (instead of trapdoor one-way functions as

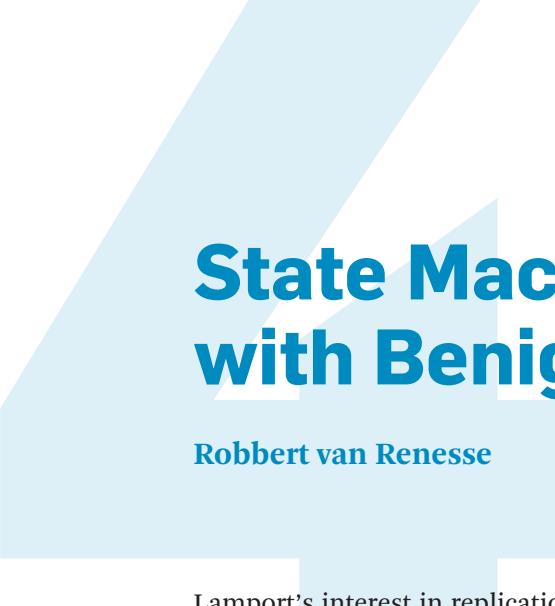
postulated by Diffie and Hellman). These constructions rely at their core on Lamport's one-time scheme, but they had no impact on the practice of cryptography.

Lamport's and Merkle's signature schemes were largely neglected until around 2005–2010 when the approach was rediscovered for practical use due to the threat that quantum computers pose to number-theoretic cryptographic primitives. All cryptographic digital signature schemes used in practice today (2019) rely on the hardness of factoring, the RSA problem, or the difficulty of computing discrete logarithms. However, a working quantum computer would make it possible to break these assumptions, as has been known since the 1990s. For these reasons, research on *quantum-safe* or *post-quantum* cryptosystems has grown tremendously in the last 15 years and become an important and successful development in cryptography.

As the security of ordinary cryptographic hash functions is not threatened, in principle, by quantum computers, research into signature schemes based on hash functions has resumed and led to many new schemes, such as XMSS [[Buchmann et al. 2011](#)], which introduced forward security to hash-based signatures. The SPHINCS scheme of [Bernstein et al. \[2015\]](#) was the first practical scheme of this kind that removed the need for the signer to store state. It relies on algorithmic advances, which can be seen to improve the efficiency of earlier theoretical constructions of suitable Merkle trees, and on a modern interpretation of what is considered to be “practical.” Due to advances in computer hardware, the signature size of SPHINCS, which amounts to some dozen kilobytes, is no longer prohibitive.

A multiyear standardization process for post-quantum cryptography has been initiated by NIST, the US National Institute of Standards and Technology [[NIST 2018](#)]; it received more than 80 submissions in late 2017. The newly proposed constructions rely on lattices, multivariate polynomials, error-correcting codes, and other primitives. Two stateless hash-based signature constructions have also been submitted, both of which extend SPHINCS. The agreement and process driven by NIST will proceed in multiple rounds and is expected to conclude with draft standards around 2022–2024.

Through these surprising developments, Lamport's work on digital signatures may also gain a lasting impact on the cryptography that secures communications over the Internet.



State Machine Replication with Benign Failures

Robbert van Renesse

Lamport's interest in replication emerged in an era when it was conjectured that no solution can keep the copies of a database synchronized, but nothing had been proved or disproved.

The earliest known examples of replication are for databases in the early 1970s (for example, [Mullery \[1971\]](#)). RFC 677 by [Johnson and Thomas \[1975\]](#) listed increasing reliability and ensuring efficiency of data access as primary motivations for replication. The RFC presented a replication technique using local timestamps, ties broken by replica identifiers, where “the latest update wins.” It may be the first example of an “eventually consistent” database, which as a concept became very popular at the beginning of the cloud computing era [[Vogels 2009](#)]. Interestingly, RFC 677 concluded that “the probability of seemingly strange behavior can be made very small. However, the distributed nature of the system dictates that this probability can never be zero.”

In 1976, [Alsberg and Day \[1976\]](#) presented a technique that goes a long way toward proving this assertion wrong, but not quite. Often claimed to be the first description of a primary-backup protocol, the basic technique involves a primary server and a backup server. Clients send update requests to the primary server, which orders and applies all incoming updates and forwards them to the backup server. The backup server also applies the updates, in the order set by the primary server, and sends acknowledgments back to the client. The paper gives no information on how to perform read commands and has only vague descriptions of how to deal with network partitions or how to generalize to multiple backups. But for the two-server case the protocol can perhaps be claimed as the first instantiation of a replication protocol that provides sequential consistency. The protocol does not

deal with mistaken failure suspicions due to, say, network partitions. Moreover, reading from the primary can lead to incorrect results. An inefficient workaround is to treat read commands the same as updates. A better solution, suggested by [van Renesse and Schneider \[2004\]](#) in a variant called *chain replication*, is to have only the backup server respond to read commands.

In 1977, Clarence Ellis of Xerox PARC developed techniques for maintaining multiple copies of a database complete with a formal specification and verification based on Petri nets [[Ellis 1977](#)]. The techniques provide eventual consistency similar to RFC 677. Again, the techniques do not address query commands, and although fault tolerance is claimed as a useful property, the solutions presented do not appear to offer it by today's understanding of fault tolerance. Specifically, for liveness the techniques require that each crashed replica is eventually restarted, and from the state at which it crashed.

Lamport's seminal 1978 "Time, Clocks, and the Ordering of Events" paper is the first to present *state machine replication* (SMR), although not by that name, as a general principle for keeping copies of a deterministic state machine synchronized, although only focusing on the failure-free case. Lamport later writes that RFC 677 inspired this paper. Indeed, Lamport proves the assertion in the conclusion of RFC 677 wrong. (As pointed out in Chapter 2, the problem with RFC 677 timestamps is that they do not capture causality.) The SMR technique is presented almost as a side note, noting its generality but developing it for the specific case of distributed mutual exclusion. However, it should be recognized as a groundbreaking result, demonstrating that, at least in the absence of failures, strong consistency (single-copy semantics) and replication can coexist.

In the same year, Lamport published "The Implementation of Reliable Distributed Multiprocess Systems" in *Computer Networks* [[Lamport 1978](#)] (first published as a Massachusetts Computer Associates Technical Report in 1977), which demonstrates protocols for SMR for synchronous networks: one for fail-stop failures and one for Byzantine failures. This may be the first paper that describes a fault-tolerant SMR protocol providing strong consistency and the first paper that describes Byzantine failures (although again not by that name).

These two Lamport papers started a cottage industry of SMR papers that continues to this day, over 40 years later. This chapter focuses on the SMR model in the face of so-called benign process failures. "Benign" is a misnomer, but it indicates that a process follows its specification until it stops altogether. In particular, a process will not execute steps that are not according to its specification, but a faulty process may stop performing steps that are in its specification.

4.1

Active versus Passive Replication

In the literature, we find a distinction between active and passive replication protocols. Informally, in active replication protocols each replica is a copy of the original state machine, starting in the same state, and each (surviving) replica is presented with the same commands in the same order. In passive replication, at any time there is only a single copy of the state machine, the primary, while the remaining replicas, the backups, only maintain state. Clients send commands to the primary. Commands that cause a state change result in the primary broadcasting “state update commands” to the backups. Such state update commands simply request overwriting part of the state kept on the backups with a specific new value. Should the primary fail, one of the backups is promoted to be primary. Note that, under this definition, the [Alsberg and Day \[1976\]](#) protocol, while often claimed to be the first passive replication protocol, is actually an active replication protocol, as all replicas are “active,” maintaining the full state machine and applying client commands in the same order.

Both active and passive replication are generic approaches to making a state machine fault tolerant. Passive replication appears more efficient as on average the replicas do less work, but failure recovery takes time; in contrast, active replication masks failures. However, a slight refactoring of function reveals that they are both instantiations of the SMR approach. In passive replication, client update requests are preprocessed and turned into state update commands. After that, all state update commands are applied to all replicas in the same order. Note that because state update commands may write overlapping parts of the state, the order is important. So for the purposes of this chapter, we do not equate active replication and SMR, but consider both active and passive replication instantiations of the SMR approach.

4.2

A Brief Review of State Machine Replication

The objective of SMR is to provide the illusion of a single “state machine,” however one that is particularly reliable and responsive in the face of various types of failures. Consider a state machine (SM) to be a process that receives a sequence of commands and produces a sequence of output values. Most importantly, the entire sequence of output values generated by an SM is completely determined by the sequence of commands the SM has processed. This appears to require that an SM is purely sequential and deterministic. Fortunately, there exist deterministic parallelism techniques that still allow an SM to leverage concurrency

[Bocchino et al. 2009]. In addition, nondeterministic choices such as reading a clock or generating a random number can be modeled as inputs to a deterministic SM.

The output values of the SM are tagged with their intended destination, often clients who send request commands to the SM. We do not, however, require that clients and SMs follow a client-server or RPC-style communication pattern. For example, one client can send a command to the SM, and the SM in response can produce multiple different output values intended for multiple other destination processes.

Both SMs and clients are instances of processes. Consistent with the end-to-end model of communication [Saltzer et al. 1984], we assume that the network that connects these processes can lose, reorder, and duplicate messages. We assume that messages have checksums so that garbled messages can be equated with lost messages. Finally, we assume that the network is fair, meaning that if a message is continuously retransmitted from a correct sending process to a correct destination process, then eventually the destination process will receive and process at least one copy of the message. (A *correct process* is a process that never crashes.) Using sequence numbers and retransmission, it is then possible to implement reliable transmission between correct processes while filtering out duplicates. We therefore assume that communication between correct processes is reliable and ordered, although it will not always be necessary to make it so. We also assume that if a process p receives a message, then some process sent that message to p .

Processes can crash, and in particular a single SM fails if the process that runs the SM crashes. To improve fault tolerance of an SM, a *replicated state machine* consists of a set of copies of an SM, aka replicas, with the following properties:

Same Initial State. The replicas start in the same state.

Agreement. If replica R_1 has processed a sequence of commands S_1 , and replica R_2 has processed a sequence of commands S_2 , then S_1 is a prefix of S_2 or vice versa.

Completion. At least one replica is correct and eventually processes all commands sent to the SM by correct clients.

For more details, we refer to Schneider's "Implementing Fault-Tolerant Services Using the State-Machine Approach" [Schneider 1990]. It is up to a specific SMR protocol to ensure that these properties hold. Given that they do, it is easy to see that the SMR produces the same output values that an unreplicated SM would have. Moreover, the duplication of output messages produced due to having multiple

replicas is filtered out by the destination processes by the very same sequence numbers used to ensure reliable end-to-end communication channels.

4.3

Benign System Models

Before we delve into SMR protocols, it is useful to look more carefully at timing properties and process failure properties. These are assumptions that one makes about the system, and it should be carefully considered that every assumption made is a potential weakness. The two extreme timing models are *synchrony* and *asynchrony*:

Synchrony. There is a known upper bound on message latencies, the time for a process to handle a command and produce zero or more output values, the drift of the clock of a process (the rate at which the clock of a process differs from real time), as well as the maximum skew (difference) between clock values of different processes at the same time.

Asynchrony. There are no assumptions on timing whatsoever.

In between, there are useful so-called semi-synchronous models. We use the following semi-synchronous model below:

Almost-Asynchrony. (aka *partial synchrony*) There is an unknown upper bound on message latencies, message handling (processing step) latencies, and clock drift.

We note that this almost-asynchronous assumption makes very weak timing assumptions. For example, it includes systems where a message between correct processes is delivered within a million years, and clocks drift by a factor of 1000 (1000 times slower or faster than real time), although processes do not actually know what these bounds are. But, from a theoretical point of view, the distinction from asynchrony is significant, as we shall soon see.

In addition, we will be more precise about various classes of benign process failures. We distinguish the following two types of assumptions about process failures:

Crash Failures. Processes follow their specification in that they do not take any steps that deviate from the specification. However, if a process crashes, then the process stops making steps indefinitely. Note that crash failures are a stronger assumption than Byzantine failures. Also note that in a synchronous environment crash failures can be detected accurately with a pinging

protocol, while no such protocol is possible in an asynchronous environment. Even the almost-asynchronous environment does not support accurate failure detection because processes do not know the timing bounds in the system.

Fail-Stop Failures. These are like crash failures, except that we assume that failures can be detected accurately by a *failure detection oracle* or simply *failure detector*. In particular, we assume that the crash of a process is eventually detected by all correct processes, while no correct process is ever suspected of having crashed. Note that fail-stop failures are essentially a synchrony assumption in disguise—in a synchronous environment we do not need to make a distinction between crash failures and fail-stop failures. But in an almost-asynchronous or asynchronous environment, fail-stop failures are a stronger assumption than crash failures.

A process that keeps its state on disk can survive power failures, but not disk failures. In fact, in an asynchronous or almost-asynchronous environment, a power failure is not a failure at all unless it persists. Thus while in practice keeping state on disk can be helpful, from a theoretical perspective it does not have much influence on the design of the protocol.

4.4

SMR Protocol Basics

An SMR protocol implements the SMR properties above given a certain set of system assumptions. There are two general—and essentially equivalent—approaches, both involving a group of participants:

Total Ordering Protocols. A total ordering protocol is a protocol in which processes broadcast messages to one another, satisfying the following properties:

T-Completion. A message sent or delivered by a correct process is eventually delivered to all correct processes.

T-Validity. If a process delivers a message, it was sent by some process.

T-Agreement. If two processes deliver the same two messages, they deliver them in the same order.

Consensus Protocols. A consensus protocol is a protocol in which processes in the group can each propose a value. The properties are then as follows:

C-Completion. If a correct process proposes or decides a value, all correct processes eventually decide a value.

C-Validity. If a process decides a value, then that value was proposed by some process.

C-Agreement. If two processes decide a value, they decide the same value.

A total ordering protocol only needs to be instantiated once per SMR. Instead, a consensus protocol is instantiated for each “slot” in the sequence of commands to an SMR. The processes can be the same set of processes that run the replicas of the SMR or a separate set of processes that relate their decisions to the replicas. It is straightforward to see that one can build a total ordering protocol out of a consensus protocol and vice versa.

The aforementioned paper “The Implementation of Reliable Distributed Multi-process Systems” by Lamport demonstrates what is possibly the first total ordering protocol for a synchronous environment with process failures. The conceptual idea is to assign a unique timestamp (ties broken by source identifier) to each message sent, and then flood the message to all other processes so all messages are delivered within a maximum amount of time that can be computed from the network topology. After this time, a message can be delivered in order of its timestamp, knowing that all prior messages must have been received as well.

The synchronous model, while simplifying the SMR problem significantly, is unfortunately rarely realistic in today’s computing environments. On the other hand, [Fischer et al. \[1985\]](#) show that in an asynchronous environment it is impossible to solve consensus in the presence of crash failures. Most protocols therefore provide only Agreement in purely asynchronous settings, and make one of the following trade-offs for Completion:

- Completion guaranteed only if the environment is almost-asynchronous.
- Completion guaranteed with probability 1 (aka Almost Surely). That is, Completion is ensured with the same probability as the probability of heads coming up when a coin is flipped indefinitely.

Both of these are eminently practical trade-offs, and thus for the remainder of this chapter we will ignore synchronous SMR protocols as well as protocols that assume fail-stop failures. Also, we will mostly cover consensus protocols rather than totally ordered broadcast protocols, because most of both the early and more recent work on SMRs has focused on the problem of consensus.

Asynchronous consensus protocols with crash failures require at least $2f + 1$ participants. To see why this is so, consider a fictitious protocol that survives f crash failures but requires only $2f$ processes. Split the processes into two groups

of f processes, one in which all processes propose some proposal Red and another in which all processes propose some proposal Blue. Consider the following three scenarios:

1. All the processes that proposed Red crash before sending any messages. The processes that proposed Blue cannot discover if the other processes proposed Red or Blue. Thus they have no choice but to decide eventually (C-Completion), and they must decide Blue (C-Validity). Assume that they do so by some time T_b .
2. Same but vice versa: The Red processes decide Red by some time T_r .
3. All processes are correct. Let T be the maximum of T_b and T_r , and assume that until time T there exists a network partition in which processes in different groups cannot exchange messages. Because the network is fair, the network partition cannot persist indefinitely. Unfortunately, to the Red and Blue processes this situation is indistinguishable from the first two scenarios, and thus they decide before T inconsistently, violating C-Agreement.

4.5

Early Asynchronous Consensus Protocols

Working up to Lamport's seminal Paxos protocol, we now review two works that precede Paxos but that illustrate various important concepts in asynchronous consensus protocols, including *rounds*, *phases*, *majority voting*, and *leaders*.

4.5.1 Ben-Or

In 1983, Michael Ben-Or published an extended abstract in the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC) called “Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols” [Ben-Or 1983]. The consensus protocol (which we will name after Ben-Or) is intended for an asynchronous environment with crash failures and guarantees both C-Validity and C-Agreement, and C-Completion with probability 1.

The protocol is binary in that there are only two possible proposals, which we call Red and Blue. Nonetheless, the protocol illustrates important concepts that most asynchronous and almost-asynchronous consensus protocols share. Assume there are N processes total, $N > 2f$. Figure 4.1 shows the protocol that each process p runs (slightly modified from its original to be more consistent with the protocols that follow in this chapter).

The protocol is an excellent illustration of Lamport's lessons on “Time, Clocks, and the Ordering of Events” in practice. In the asynchronous model, there is no

0. Initially, set estimate e_p to p 's proposal and round number $r_p = 1$.
1. Phase P1:
 - (a) Broadcast message $\langle phase = P1, round = r_p, estimate = e_p \rangle$.
 - (b) Wait for messages of the form $\langle phase = P1, round = r_p, estimate = *\rangle$ from $> N/2$ different processes.
 - (c) If all contain the same estimate e , then set $v_p = e$. Otherwise set $v_p = \perp$.
2. Phase P2:
 - (a) Broadcast message $\langle phase = P2, round = r_p, estimate = v_p \rangle$.
 - (b) Wait for messages of the form $\langle phase = P2, round = r_p, estimate = *\rangle$ from $> N/2$ different processes.
 - (c) (i) If all messages contain the same non- \perp estimate e , then **decide** e .
 - (ii) If one of the messages contains a non- \perp estimate e , then **accept** e by setting $e_p = e$. Otherwise **accept** a random estimate by setting e_p to Red or Blue uniformly at random.
3. Increment r_p and go to Step 1.

Figure 4.1 A variant of the Ben-Or consensus protocol.

concept of time or clocks. The round number r_p simulates a local clock for p . However, note that there is no bound on the difference between the round numbers of two different processes, nor on the rate at which the round number of a process increases. Process p only considers messages for the round r_p that it is in; it ignores messages for prior rounds and buffers messages for future rounds. Doing so, the processes can pretend that together they run one round at a time, but in reality their execution may be arbitrarily skewed and different rounds will typically overlap in real time.

Each process p maintains an estimate e_p . A process can accept as well as decide an estimate in the second phase of a round. Accepting an estimate simply means that a process updates its own estimate. (Accepting is also referred to as *voting*, but we use Lamport's terminology here.) It is guaranteed that if a process decides estimate e , it also accepts e .

Define an estimate e to be *safe* in round r if the following two properties hold:

1. Some process proposed e as its initial estimate in step 0; and
2. No process decides or has decided a different estimate in round r' where $r' \leq r$.

An important invariant to keep in mind in this consensus protocol is that at any time and for any process p , any estimate that p accepts is safe in round r_p . A similar invariant usually holds in other asynchronous consensus protocols. Because a process never decides an estimate that it does not accept in the same round, C-Validity and C-Agreement follow trivially from this invariant.

The protocol subdivides a round into two subrounds or *phases*, P1 and P2. Each phase consists of each correct process p performing three actions:

- (a) p broadcasts a message (sent to all processes including self);
- (b) p waits for more than $N/2$ messages (but no more than $N - f$) from peers (including self);
- (c) p updates its local state based on its current state and the messages that p has received.

Inductively, it is easy to show that each phase, and therefore each round, eventually terminates. After all, given that no more than f processes crash and $N > 2f$, each process is guaranteed to receive $N - f$ ($N - f > N/2$) messages in each phase.

An invariant of the Ben-Or protocol is that the estimates of all processes going into phase P1 (either from step 0 or from step 3) of round r are safe in round $r - 1$. It is clear that this is true for the initial estimates (all estimates are safe in round 0 as no proposal has been decided yet), and we will informally argue that it also holds for later rounds while explaining the protocol.

The purpose of phase P1 in the Ben-Or protocol is to reduce the number of estimates that are eligible for decision in phase P2 to at most a single one. To win, each process p computes a value v_p that is either a proposal (Red or Blue) or \perp in step 1c after having received a majority of estimates. Because it is not possible that there exist two different majorities, one in which all processes have sent estimate Red and another in which all processes have sent estimate Blue, it must be the case that if two processes p and q compute non- \perp values v_p and v_q , then $v_p = v_q$. It is also important to note that, if all processes that broadcast an estimate in step 1a broadcast the same estimate e , then all processes p that set v_p in step 1c set $v_p = e$.

The purpose of phase P2 is for each process p to possibly decide an estimate and in any case accept an estimate that is safe in round r_p . A process p decides an estimate e in round r_p if it receives that estimate from more than a majority of processes in phase P2 of round r_p .

First assume this is the case. Because all other processes wait for P2 messages from a majority of processes, it follows that all other processes that reach step 2c in the same round must receive at least one P2 message with the same estimate

e . Moreover, as we saw above, it is not possible that processes receive a non- \perp estimate other than e . Thus all processes that reach step 2c of round r will accept (and possibly decide) e in round r . Moreover, as e was safe in round $r - 1$ and no estimate other than e could have been decided before round r , e must still be safe. In this case, all processes that enter the next round $r + 1$ enter with their estimate set to e , satisfying the invariant.

Perhaps the more interesting case is what happens when no process decides in some round r because no process received a non- \perp estimate from a majority. Note that if some estimate e had been decided before round r , then only that estimate would have been safe in round $r - 1$ and all processes would have received e in step 1c. Since this is not the case, we may conclude that no estimate was decided before round r . Because no process decides in round r either, both Red and Blue are safe estimates in round r .

In this case, some processes may still receive a non- \perp estimate in step 2b. Recall that all such processes are guaranteed to receive the same estimate e and thus will set their estimate to e in step 2c.ii. The remaining processes accept a random estimate in step 2c.ii. By accepting a random estimate if only \perp messages are received in phase P2, there is a chance, even if small, that all processes doing so happen to accept the same estimate e in step 2c.ii. And this in turn would cause all correct processes to decide e in the next round. Thus each round (except possibly the first) has a nonzero chance of deciding. Given that the number of rounds is unbounded, the Ben-Or protocol satisfies C-Completion with probability 1, just like the probability of heads coming up is 1 if a coin is flipped an unbounded number of times. This convergence can be sped up dramatically if the processes use a so-called shared coin, that is, a pseudorandom coin that comes up the same at all processes in the same round.

Note that although the protocol as described continues to run even after all correct processes have decided, it is easy to see that a process that decides can stop running the protocol after finishing just one more round, as all correct processes are guaranteed to have decided by then.

The reader may wonder why it is necessary to have two phases, rather than just one. In fact, it is easy to design a consensus protocol that has only one phase. However, the minimal number of participants for such a protocol has to be $3f + 1$ in order to tolerate f failures [Brasileiro et al. 2001]. By reducing the number of eligible proposals to at most one in phase P1 of the Ben-Or protocol, a simple majority vote in phase P2 is sufficient to decide. Replicas are expensive, and two phases allow asynchronous consensus protocols to meet the $2f + 1$ lower bound on the number of replicas.

4.5.2 Dwork, Lynch, and Stockmeyer

In 1984, one year after the publication of the Ben-Or consensus protocol, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer published a preliminary version of their “Consensus in the Presence of Partial Synchrony” paper [Dwork et al. 1988] about a consensus protocol (referred to here as the DLS protocol after its authors) for the almost-synchronous model that illustrates two other important concepts. The first is the concept of using a leader, and the second is demonstrating that consensus can satisfy C-Completion in the almost-asynchronous model.

As pointed out above, asynchronous consensus protocols that meet the lower bound on the number of replicas need two phases, the first of which reduces the number of proposals eligible for decision and acceptance to at most one. For the Ben-Or protocol, this was accomplished by a majority vote in the first phase. A simpler approach is to assign to each round a leader and have that leader select a safe estimate for that round. Should the leader crash, then the next round, with another leader, can still make progress.

We will not reproduce the entire DLS protocol here. Instead, we modify the Ben-Or protocol to use a leader in phase P1. We call the resulting protocol *Franken consensus*. Let the leader of a round r be the participating process $r \bmod N$. Figure 4.2 shows the steps that each process p executes.

There are three important differences from the Ben-Or protocol:

1. The Franken protocol uses a timeout parameter T_p , which is initialized to a nonzero value and increased for each round (doubling is not essential, as long as T_p increases by a nontrivial amount on each round).
2. Phase P1 of the protocol has a leader proposing a particular estimate instead of the processes determining one by majority vote. Note that the number of messages that cross the network in phase P1 is reduced from $N(N - 1)$ to $N - 1$. If a process times out waiting for the estimate from the leader of the current round, it continues with the \perp value.
3. In step 2c.ii, if a process receives only messages containing \perp estimates, then it accepts its current estimate, rather than accepting a random estimate.

Phase P1 of Ben-Or consensus and Franken consensus are different implementations but serve exactly the same purpose: to select at most one of the estimates that are safe in round $r - 1$ for acceptance by the processes in round r . Initially, all processes may time out, but eventually parameter T_p will grow sufficiently large so that other processes will not time out waiting for messages from a correct leader, guaranteeing C-Completion. However, until then it is possible that all processes do

0. Initially, set estimate e_p to p 's proposal, round number r_p to 1, and T_p to some initial value > 0 .
1. Phase P1:
 - (a) If p is leader of r_p , broadcast $\langle phase = P1, round = r_p, estimate = e_p \rangle$.
 - (b) Leader or not:
 - (i) Wait for a message of the form $\langle phase = P1, round = r_p, estimate = e \rangle$ from the leader of r_p , up to T_p ticks measured on p 's clock.
 - (ii) Upon receipt set $v_p = e$. On timeout, set $v_p = \perp$.
2. Phase P2:
 - (a) Broadcast $\langle phase = P2, round = r_p, estimate = v_p \rangle$.
 - (b) Wait for messages of the form $\langle phase = P2, round = r_p, estimate = * \rangle$ from $> N/2$ different processes.
 - (c)
 - (i) If all messages contain the same non- \perp estimate e , then **decide** e .
 - (ii) If one of the messages contains a non- \perp estimate e , then **accept** e by setting $e_p = e$. Otherwise **accept** e_p (i.e., leave e_p unchanged).
3. Increment r_p , double T_p , and go to step 1.

Figure 4.2 The Franken consensus protocol.

timeout waiting for a correct leader *even if all estimates started out the same*. Thus, unlike in the Ben-Or protocol, a process may not decide in phase P2 if all estimates going into phase P1 are the same, and in particular, a process may not decide in round r if some estimate was decided in a round prior to r .

In step 2c, it would not be safe for a process to accept a random estimate if it received only \perp estimates. Instead, a process accepts its current estimate. To see why this estimate is safe, consider the following. As in the Ben-Or protocol, if a process decides an estimate e in a round r , all correct processes are guaranteed to accept (and possibly decide) e in round r . Moreover, going into round $r + 1$ the only estimate left in play is estimate e because processes running the Franken protocol accept no random values.

If the initial timeout parameter of each process is chosen reasonably well and crash failures are rare, Franken consensus is a significant improvement in practice over Ben-Or consensus. In this sometimes called “normal case,” all correct participants in the Franken consensus protocol will decide the leader’s estimate in a single round, even if all processes start with different proposals. Also note that, unlike the Ben-Or protocol, Franken consensus supports more than two different proposals.

The published DLS protocol, while different, shares all these advantages with the Franken protocol. Unfortunately, the DLS protocol remained a theoretical curiosity, and SMR protocols at the time only existed in the form of primary-backup protocols. Such protocols assume fail-stop failures and can violate Agreement when there are latency anomalies either in the network or in a process itself violating the fail-stop assumption.

4.6

Paxos

Lamport discovered the Paxos protocol in the late 1980s while investigating the fault tolerance of the Echo file system being developed at Digital's Systems Research Center [Hisgen et al. 1989]. Paxos made SMR for almost-asynchronous environments significantly more practical. Described in his seminal “The Part-Time Parliament” paper [Lamport 1998a], its underlying consensus protocol (known as the Synod protocol, but these days most use Paxos to refer to both the SMR protocol and the underlying consensus algorithm) has two important improvements over prior protocols:

1. Like other protocols that meet the $2f + 1$ lower bound, Paxos uses two phases per round (called *ballots* in Paxos), but both phases P1 and P2 only use $O(N)$ messages instead of $O(N^2)$ messages. Hence a round only uses $O(N)$ messages.
2. In the “normal case” in which failures and failure suspicions are rare, the leader of a round can make a series of decisions, requiring running only phase P2 for each command executed by the SMR.

As in the original Paxos paper, we develop the protocol in two steps: first reducing the message complexity, and then eliminating the first phase except in case a leader is suspected of having failed. For the first protocol, we reuse the structure of the Franken protocol, in the hope that the intuitions about consensus in the almost-asynchronous model learned above help with understanding the Paxos protocol. However, each phase consists of two subphases:

- (a) The leader broadcasts a message to the processes, called *acceptors* in Paxos. Upon receipt, an acceptor updates its state;
- (b) Each acceptor that received a message responds to the leader. The leader awaits responses from a majority of acceptors and updates its state.

Figure 4.3 shows the steps that each process p executes in a variant of Lamport's “single-decree Synod protocol.”

0. Initially, set estimate $e_p = \perp$, round number $r_p = 1$, and T_p to some initial value > 0 .
1. Phase P1:
 - (a) (i) If p is leader of r_p , broadcast $\langle \text{phase} = P1a, \text{round} = r_p \rangle$.
 (ii) Leader or not:
 - Wait for a message of the form $\langle \text{phase} = P1a, \text{round} = r \rangle$ such that $r \geq r_p$.
 - Set $r_p = r$.
 - (b) (i) Send $\langle \text{phase} = P1b, \text{round} = r, \text{estimate} = e_p \rangle$ to the leader of r .
 (ii) If p is leader of r_p , then:
 - Wait for messages of the form $\langle \text{phase} = P1b, \text{round} = r_p, \text{estimate} = * \rangle$ from $> N/2$ different processes.
 - If some estimates are non- \perp , set v_p to the proposal in the highest numbered estimate. If instead all estimates are \perp , then set v_p to the initial proposal of p .
2. Phase P2:
 - (a) (i) If p is leader of r_p , broadcast $\langle \text{phase} = P2a, \text{round} = r_p, \text{estimate} = \langle r_p, v_p \rangle \rangle$.
 (ii) Leader or not:
 - Wait for a message of the form $\langle \text{phase} = P2a, \text{round} = r_p, \text{estimate} = e \rangle$ from the leader of r_p .
 - **Accept** e by setting $e_p = e$.
 - (b) (i) Send $\langle \text{phase} = P2b, \text{round} = r_p \rangle$ to the leader of r_p .
 (ii) If p is leader of r_p , then:
 - Wait for messages of the form $\langle \text{phase} = P2b, \text{round} = r_p \rangle$ from $> N/2$ different processes.
 - **Decide** the proposal in e_p and broadcast decision to all replicas.
3. If p is not leader of round r_p , then upon timeout waiting for a message from the leader of round r_p :
 - Increase r_p to the next round for which p is leader.
 - Double T_p and go to step 1.

Figure 4.3 A variant of the single-decree Paxos protocol.

While the protocol resembles the Franken protocol, there are some important differences to note:

- Estimate e_p is no longer a proposal, but initially \perp and later assigned to be a $\langle \text{round number}, \text{proposal} \rangle$ pair. There cannot be two different estimates with the same round number (as the leader of a round only generates at most one).

Estimates are ordered by round number, with \perp representing the smallest possible estimate.

- In the Franken protocol, all estimates going into phase 1 of round r are safe in round $r - 1$. Not so in Paxos. In phase P1, the leader of round r does not distribute its estimate, because it may not be safe. Instead, it announces only its round number and collects the estimates of its peers, who promise not to accept more estimates in rounds less than r . After hearing back from a majority, the leader determines the highest estimate among them. If this is \perp , the leader learns that no proposal was decided in rounds lower than r . In this case, it will use as a safe estimate $\langle r, \text{leader's own proposal} \rangle$. Otherwise, the highest estimate is safe in round r .
- In phase P2, instead of an N^2 communication pattern, the leader broadcasts its estimate. If still acceptable given the peer's round number, the peer accepts the estimate and responds to the leader. If the leader hears back from a majority, it knows that any future leader that might take over will learn about this estimate, and therefore it can decide the proposal in the estimate.
- A process p no longer simply runs one round at a time. Instead, it can *skip* to any future round when it receives a P1a message from the leader of such a round. When doing this, its stored estimate in e_p may no longer be safe in the new round. This is why eligible estimates are determined at the end of phase P1, rather than at the beginning.
- A leader may crash, and therefore each wait on a message from the leader has a timeout of T_p time units. When timing out on a leader, a process tries to become the leader itself by initiating a new round, as specified in step 3.

If all this sounds complicated, there is a reason for that: it is difficult for anyone but a computer to understand an operational description of a protocol. The beauty of the “Part-Time Parliament” paper is that it starts out with demonstrating what principles a consensus protocol must satisfy in order for it to satisfy C-Validity and C-Agreement. (The paper does not address C-Completion.) As a result, the paper describes a family of consensus protocols that satisfy C-Validity and C-Agreement by construction. We will not even attempt to duplicate the beautiful presentation of the “Part-Time Parliament” paper here (or its “simpler cousin” [Lamport 2001]), for the simple reason that there is no need to.

The “Part-Time Parliament” goes on to describe an important improvement by allowing the same leader to decide a sequence of proposals rather than just a single

one. This way, execution of phase P1 can be avoided unless the leader is suspected of having crashed. This advance is made possible because a leader does not propose a value until the end of phase P1. The leader can continue to extend the sequence of decided proposals until (after a timeout causing the leader to be suspected of having crashed) a majority of acceptors have moved on with a larger round number. The resulting protocol is these days often called Multi-Paxos [[Chandra et al. 2007](#)].

4.6.1 Read-Only Commands

In many workloads, read commands are much more frequent than update commands, and thus their efficiency is important to the overall practicality of SMR. If clients were to read from any replica, they may well read from a stale one, violating one-copy semantics. Performing read commands just like update commands is inefficient, particularly if Paxos state is kept on disk. Reading from all replicas and selecting the most recent one would not be fault tolerant and intolerably slow even if all replicas were up. It is possible to read from a quorum of replicas, but obtaining the state this way is tricky to get right and still inefficient compared to reading off a single replica.

To solve the problem, the “Part-Time Parliament” paper proposes to appoint primary replicas for specific intervals of real time. This is akin to leases [[Gray and Cheriton 1989](#)]. Only one replica can have a lease at a time. Only a replica with a valid lease can produce output messages, and only that replica can respond to queries. The time intervals assigned to leases should be short so that, should the lease holder crash, it will not be long before another replica can take over. This elegant solution has approximately the same read efficiency as an unreplicated SM, but note that it requires that processor clocks are tightly synchronized. While it is possible to allow for a known bounded skew between clocks, or even just to rely on a known bounded drift of clocks, these assumptions are still stronger than the almost-asynchronous model. In practice, however, it is easy to have access today to clocks with bounded skew and/or drift, while message latencies and processing times are still hard to bound.

4.6.2 Discussion

While it is often claimed that viewstamped replication (VR) by [Oki and Liskov \[1988\]](#) bears a resemblance to Multi-Paxos (and indeed, VR can be considered a protocol in the Paxos family of consensus protocols), the VR protocol is a specific implementation of a consensus protocol for passive replication. The VR protocol is a significant advance, but the paper gives only an operational description of the VR protocol and contains no clear expression of principles nor a proof of

the protocol's correctness. The “Part-Time Parliament” paper thus represents an important contribution to the science of fault-tolerant asynchronous distributed systems even if an example of a protocol in the Paxos family had been demonstrated previously.

Paxos was first deployed in the Petal distributed storage system [Lee and Thekkath 1996]. In time, Paxos has become probably the best-known replication protocol and variants of it have been widely deployed, including in systems such as Google’s Chubby system [Burrows 2006] and Microsoft’s NoSQL Azure Cosmos database, and in countless open source projects such as Yahoo!/Apache ZooKeeper [Hunt et al. 2010], Facebook Cassandra [Lakshman and Malik 2009], Ceph [Weil et al. 2006], and Raft [Ongaro and Ousterhout 2014].

Disk Paxos [Gafni and Lamport 2003] replaces acceptor processes with disks that support only read block and write block operations and can therefore be considered a “shared memory” version of Paxos. It was originally developed in 1998 for an interesting scenario at Digital Equipment Corporation’s storage group, in which there were only two processors. Two processors are generally not enough to tolerate a crash failure in an asynchronous environment, but it turns out it can be done if there are at least three independent disks that can be accessed by both processors. In general, Disk Paxos requires $m > f$ processors and $n > 2f$ disks to tolerate f processor and f disk failures.

The protocol still resembles the original Paxos. There are m leaders, one on each processor. Each leader has a block on each disk that only that leader can write, but all leaders can read all blocks. Thus there are a total of $n \times m$ blocks. To start a new round, a leader remotely writes a P1a message to its block on at least a majority of the disks, simulating a broadcast to the acceptors. The leader then tries to read all m blocks from each of a majority of disks to see if some other leader initiated a higher round. If not, the leader moves on to phase P2, which proceeds similarly.

4.7

Dynamic Reconfiguration

Being able to replace replicas is important for fault-tolerant services that must remain up 24/7 and cannot be taken down temporarily for maintenance operations, such as replacing failed replicas or performing hardware upgrades. Lamport described how to do this in synchronous environments in his 1984 paper “Using Time Instead of Timeout for Fault-Tolerant Distributed Systems.” The basic idea is to introduce a reconfiguration command that replaces the set of replicas. The “Part-Time Parliament” paper adopts the same approach, but using slot numbers

(indexes into the message input stream to the SMR) instead of time. So slot i could contain a reconfiguration command that specifies what the configuration is starting at slot $i + \alpha$. While α could be as small as 1, this would require that one cannot know the configuration for slot $i + 1$ until after a command is decided for slot i . Hence no concurrency would be possible. For $\alpha > 1$, the current configuration of replicas can be deciding α commands (slots i through $i + \alpha - 1$) in parallel, which can improve throughput.

While perhaps sounding simple enough, dealing with reconfiguration has been an Achilles' heel for practical Paxos deployments. The dynamic reconfiguration approach described above has complicated implementation corner cases. For example, clients have to deal with the fact that multiple configurations may be simultaneously active, while replicas have to deal with the fact that decisions may come out of order, both within a configuration and across configurations. Also, as multiple (α) decisions may be outstanding at any time, a single configuration can, in theory, decide on multiple future configurations.

Much work in the last couple of decades has gone into refining failure handling and recovery for practical, large-scale deployments of Paxos. An early approach, Cheap Paxos [Lamport and Massa 2004], looks at reducing cost in the normal case while still being able to reconfigure the replicas should the need arise. Instead of using $2f + 1$ or more acceptors, Cheap Paxos uses only $f + 1$ acceptors combined with f “auxiliary” acceptors that are idle during normal execution but play an important role in failure recovery. The optimization is based on the observation that during normal operation Paxos only needs a majority of processes to participate in the protocol. However, if one of those processes becomes unresponsive, some of the auxiliary acceptors are necessary to run phase P1 of the protocol. The reconfiguration protocol of Paxos can then be used to replace the unresponsive acceptors with new ones, restoring the fault tolerance of the system.

Vertical Paxos [Lamport et al. 2009a] takes this idea a step further, fixing the round number and leader during normal operation, similar to primary-backup protocols. When a failure occurs, an auxiliary “configuration master” decides on what the new configuration will be. The new leader then accesses the state of the processes in the old configuration to obtain the old state and informs the configuration master. This old state can then be used to create a new configuration that is a consistent continuation of the old one. A similar technique is used for recovery in the Google File System [Ghemawat et al. 2003] and the chain replication protocol [van Renesse and Schneider 2004]. In order to be fault tolerant, the configuration master itself must be replicated and may also require dynamic reconfiguration.

An alternative approach to dynamic reconfiguration is to, in a single reconfiguration operation, terminate the old configuration and create a new one. This approach takes inspiration from so-called view changes in group communication protocols such as Isis [Birman et al. 1991]. For example, in Stoppable Paxos [Lamport et al. 2010], a special STOP reconfiguration command has the simplifying property that if some configuration decides a STOP command in slot i , then that configuration can no longer decide commands in slots after i (while still allowing concurrency). In Lamport et al. [2010], Lamport, Malkhi, and Zhou discuss advantages and disadvantages of various SMR reconfiguration methods.

Formal Specification and Verification

Stephan Merz

5.1

Introduction

Beyond his seminal contributions to the theory and the design of concurrent and distributed algorithms, Leslie Lamport has throughout his career worked on methods and formalisms for rigorously establishing the correctness of algorithms. Commenting on his first article about a method for proving the correctness of multiprocess programs [Lamport 1977a] on the website providing access to his collected writings [Lamport 2019], Lamport recalls that this interest originated in his submitting a flawed mutual exclusion algorithm in the early 1970s. As a trained mathematician, Lamport is perfectly familiar with mathematical set theory, the standard formal foundation of classical mathematics. His career in industrial research environments and the fact that his main interest has been in algorithms, not formalisms, has certainly contributed to his designing reasoning methods that combine pragmatism and mathematical elegance. The methods that he designed have always been grounded in the semantics of programs and their executions rather than being driven by the syntax in which these programs are expressed, in contrast to many methods advocated by “pure” computer scientists.

The famous “Time/Clocks” paper [Lamport 1978b] describes executions of distributed state machines and introduces the happened-before or causality relation, a partial order on the events occurring in runs. The “philosophically correct” way for reasoning about distributed executions would thus appear to be based on a partial ordering between operations. Indeed, Lamport explored this idea and proposed a method based on two relations called *precedes* and *may affect* [Lamport 1979c]. This method can notably be applied to algorithms that involve nonatomic operations, such as the Bakery algorithm. However, Lamport felt that the method did not scale

well, unlike assertional reasoning about global states of systems that can be visible to an idealized external observer, even if no single process can observe them. This style of reasoning considers linearizations of distributed executions, and it generally requires algorithms to be described in terms of their atomic operations (an exception being [Lamport 1990]). The notion of an overall system invariant that is preserved by every operation plays a central role in this approach: such an invariant explains why the algorithm is correct. Assertional proofs have since been demonstrated to be completely rigorous, to be amenable to mechanized checking, and to scale well.

Lamport realized that there are two fundamental classes of correctness properties that arise in the verification of concurrent algorithms, for which he coined the terms *safety* and *liveness* properties [Lamport 1977a]. Generalizing, respectively, partial correctness and termination of sequential programs, safety properties assert that “nothing bad ever happens” and liveness properties require that “something good happens eventually.” These intuitive concepts were later formalized by Alpern and Schneider [1985], who showed that any property of runs can be expressed as the intersection of a safety property and a liveness property. Although proofs of liveness properties generally rely on auxiliary invariants, the basic principles for proving safety and liveness properties are different, and the two are therefore best considered separately.

Lamport advocates describing algorithms in terms of state machines whose operations are atomic. Invariants, and more generally safety properties, are then established by induction: the invariant holds in all possible initial states and is preserved by every operation of the state machine. The proof of liveness properties usually relies on associating a measure with the states of the algorithm and showing that the measure decreases with every step as long as the “good state” has not been reached. This argument is made formal through the use of well-founded orderings, which do not admit infinite decreasing chains. A direct application of that proof principle would require fixing a scheduler that governs the execution of different processes—an undesirable requirement since one wants to establish the correctness of the algorithm for any “reasonable” scheduler. A useful generalization requires that as long as the target state has not been reached, no step of the algorithm increases the measure, “helpful” steps decrease the measure, and some helpful step will eventually be executed. In order to justify the latter (which in itself is a liveness property!), one invokes fairness assumptions [Attie et al. 1993, Lamport 2000] that assert that executable operations will not be neglected forever, fixing the precise understanding of a “reasonable” scheduler for the particular application.

Another fundamental concept underlying the rigorous development and proof of concurrent algorithms advocated by Lamport is that of *refinement*. It allows a

designer to describe the fundamental correctness properties using a high-level (possibly centralized) state machine and then prove that another state machine whose description is given at a lower level of abstraction faithfully implements the high-level description. For example, a high-level state machine describing a consensus algorithm [Lamport et al. 1982] could have a variable *chosen* holding a set of values, initialized to the empty set, a *Choose* operation that assigns *chosen* to the singleton set $\{v\}$ for some value v among the proposed values, and *Decide* operations that set the decision values of each process to that chosen value. A lower-level refinement would then describe the actual algorithm in terms of exchanged messages and votes. A technical complication in that approach is that the lower-level state machine will have operations that modify only low-level variables, i.e., variables that do not exist at the higher level of abstraction. These operations cannot be mapped to operations of the high-level state machine. For example, operations that send messages in the putative consensus algorithm have no meaning in the high-level specification. Lamport advocates that formalisms for describing executions of state machines should be insensitive to *stuttering steps* that leave unchanged the state visible to the specification.

The remainder of this chapter focuses on the Temporal Logic of Actions (TLA) and the specification language TLA⁺. These are Lamport's contributions to the formal specification and verification of algorithms that have had the greatest impact in the academic community and in industry, and their design was guided by the principles that have been outlined above.

5.2

The Temporal Logic of Actions

Temporal logic [Prior 1967] is a branch of modal logic in which the accessibility relation corresponds to temporal succession. During the 1970s, several authors [Burstall 1974, Kröger 1977] suggested adopting temporal logic as a basis for proving programs correct. Pnueli's influential paper [Pnueli 1977] provided the insight that temporal logic was most useful for specifying and reasoning about *reactive systems*, which include concurrent and distributed systems. A generalization of Pnueli's logic that is based on the “next” and “until” connectives remains the standard variant of linear-time temporal logic used in computer science to this day.

5.2.1 The Genesis of TLA

In 1977, Lamport proposed a method for proving the correctness of concurrent programs [Lamport 1977a]. It used invariant reasoning (based on Floyd's method [Floyd 1967]) for establishing safety properties and lattices of leads-to properties

for proving liveness. The method relied on a fixed progress assumption for each process in order to establish elementary leads-to properties.

Lamport was introduced to temporal logic in the late 1970s during a seminar organized by Susan Owicky at Stanford University. At that time, the distinction between linear-time and branching-time temporal logics was not yet clearly established in computer science. Lamport clarified this difference [Lamport 1980] and showed that the expressive powers of LTL and CTL are incomparable. He quickly realized that temporal logic was a convenient language for expressing and reasoning about fairness and liveness properties. For example, weak and strong fairness of executions with respect to an action α , representing an operation of a process in a concurrent system, can be written as

$$\square(\square en(\alpha) \Rightarrow \diamond exec(\alpha)) \quad \text{and} \quad \square(\square \diamond en(\alpha) \Rightarrow \diamond exec(\alpha)).$$

In these formulas, the predicate $en(\alpha)$ characterizes those states in which α is enabled (may be executed), $exec(\alpha)$ is true when action α has been executed, and \square and \diamond are the “always” and “eventually” operators of LTL. Weak fairness requires that an action cannot remain perpetually enabled without eventually being executed. Strong fairness requires that even an action that is infinitely often (but perhaps not perpetually) enabled must eventually be executed. Using such formulas, more general fairness hypotheses than uniform progress of processes considered in Lamport [1977a] can be expressed unambiguously. Moreover, the principles of reasoning about leads-to lattices could be derived from the general proof rules of temporal logic. A joint paper with Owicky [Owicky and Lamport 1982] develops these ideas into a full-fledged method for proving the correctness of concurrent programs. In the introduction to this paper, the authors write:

While we hope that logicians will find this work interesting, our goal is to define a method that programmers will find useful.

This motto describes well Lamport’s approach to formalisms for specification and verification.

Whereas standard LTL was clearly useful for expressing fairness and liveness properties, Lamport felt that it was not convenient for writing complete specifications of actual systems. His intuition was confirmed when he observed colleagues at SRI struggling with specifying a FIFO queue in Pnueli’s temporal logic. (It was later proved that this was actually impossible unless one assumes that the values presented to the queue are unique.) This observation led his colleagues to introduce a more expressive temporal logic based on intervals [Schwartz et al. 1984]. In contrast, Lamport concluded that the fundamental problem was the “property-

oriented" style of specifications as a list (conjunction) of properties observed at the interface of a system, such as the inputs and outputs of the queue, but excluding any reference to internal system states. He designed TLA as a logic geared toward specifying and reasoning about state machines, based on a few orthogonal and simple concepts that provide a higher level of abstraction (and elegance!) than the use of a pseudo-programming language, as in the earlier paper with Owicki [[Owicki and Lamport 1982](#)].

5.2.2 The Logic TLA

Lamport designed TLA around 1990 [[Lamport 1991](#), [1994c](#)]. TLA formulas are built from *constants*, whose values are fixed throughout an execution, and (state) *variables*. We use x, y, z to denote constants and u, v, w to denote variables. A *state* is a mapping from variables to values. TLA distinguishes three levels of expressions:

- The syntax of state functions and state predicates is that of standard terms and formulas of first-order logic. Concrete examples of state predicates are $v \geq 0$ or $\exists x : x \in u \wedge x \notin v$. Semantically, they are interpreted over individual states.¹
- Transition functions and transition predicates, also called *actions*, are first-order terms and formulas that may contain both standard (unprimed) variables u, v, w and primed variables u', v', w' . Example formulas are $u' \in v$ or $\exists x : u' + x = v$. Semantically, transition formulas are interpreted over pairs $\langle s, t \rangle$ of states, with unprimed variables being interpreted in state s and primed variables in t .
- Temporal formulas are built from state and transition formulas by applying operators of temporal logic according to the rules given below. They are interpreted over *behaviors*, i.e., sequences $\sigma = \langle s_0, s_1, \dots \rangle$ of states.

Whereas standard LTL builds temporal formulas solely from state formulas, the introduction of transition formulas as a primitive building block is fundamental to specifying state machines. For example, the LTL equivalent to the TLA action $u' \in v$ would be

$$\exists x : x \in v \wedge \circ(u = x)$$

1. We could distinguish a level of constant formulas that do not contain any variables, but we will consider such formulas to be state formulas.

where \circ denotes LTL's next-time operator. Rather than using a pseudo-programming notation as in [Owicki and Lamport \[1982\]](#), actions are just first-order formulas over primed and unprimed variables. One reason about them using ordinary mathematical logic, rather than introducing special principles for reasoning about programs. At the temporal level, any formula can in principle be considered as a system specification or as a property. There is no formal distinction between the two, and reasoning about them relies on the same fundamental principles of temporal logic.

TLA introduces several notations at the levels of state and transition formulas. Given a state formula e , the transition formula e' is obtained by replacing all (free) occurrences of state variables by their primed counterparts. Semantically, e' denotes the value of e at the second state of the pair of states at which e' is evaluated. The action UNCHANGED e is shorthand for $e' = e$. For an action A and a state formula e , the actions $[A]_e$ and $\langle A \rangle_e$ stand for $A \vee e' = e$ and $A \wedge e' \neq e$, respectively. The action formula $[A]_e$ represents closure of A under stuttering (with respect to e); in particular, it is true of a pair of states $\langle s, t \rangle$ if A is true or if $s = t$. Dually, $\langle A \rangle_e$ requires A to be true and the step from state s to state t to be observable through a change of e . Finally, the state predicate ENABLED A is obtained by existential quantification over all primed state variables that occur in the action A . For example,²

$$\begin{aligned} & \text{if } A \triangleq v > 0 \wedge v' = v - 1 \wedge w' = w \\ & \text{then } \text{ENABLED } A \triangleq \exists v', w' : v > 0 \wedge v' = v - 1 \wedge w' = w \end{aligned}$$

It is easy to see that for this example, $\text{ENABLED } A$ is logically equivalent to the predicate $v > 0$. In general, $\text{ENABLED } A$ is true at state s if there exists some state t such that A holds for the pair $\langle s, t \rangle$.

Formulas at all three levels are closed under Boolean operators ($\neg, \wedge, \vee, \Rightarrow, \equiv$) and first-order quantifiers (\forall, \exists). The rules for forming temporal formulas are as follows:

- Every state predicate is a temporal formula.
- If A is an action and e is a state formula, then $\square[A]_e$ is a temporal formula.
- If φ is a temporal formula, then so is $\square \varphi$.
- If φ is a temporal formula, x is a constant, and v is a variable, then $\exists x : \varphi$ and $\exists v : \varphi$ are temporal formulas.

2. The symbol \triangleq denotes "is defined as."

The formulas $\diamond \varphi$ and $\diamond \langle A \rangle_e$ are shorthand for $\neg \square \neg \varphi$ and $\neg \square \neg [A]_e$, respectively. Also, $\varphi \rightsquigarrow \psi$ (“ φ leadsto ψ ”) abbreviates $\square(\varphi \Rightarrow \diamond \psi)$. Observe in particular that if A is an action formula, $\square A$ is in general not well formed: actions need to be “protected” by square or angle brackets inside temporal formulas.

The operators \square and \diamond are the familiar “always” and “eventually” operators of LTL: $\square \varphi$ is true of σ if φ is true of every suffix of σ . The formula $\square [A]_e$ is true of $\sigma = \langle s_0, s_1, \dots \rangle$ if, for all $n \in \mathbb{N}$, the action A holds for the state pair $\langle s_n, s_{n+1} \rangle$ or the state formula e evaluates to the same value in s_n and s_{n+1} , and the interpretation of $\diamond \langle A \rangle_e$ is dual. The syntactic restriction of allowing action formulas to appear only inside brackets ensures that all temporal formulas φ of TLA are insensitive to finite stuttering: if two state sequences σ and τ agree up to insertions or removals of finite repetitions of states, then φ is true of σ if and only if φ is true of τ . The formula $\exists v : \varphi$ is true of $\sigma = \langle s_0, s_1, \dots \rangle$ if φ is true of a sequence $\tau = \langle t_0, t_1, \dots \rangle$ such that, for all n , s_n and t_n agree on the values of all variables except possibly v . The formal definition is somewhat more complicated in order to preserve invariance under stuttering [Lamport 1994c].

The notion of validity of TLA formulas is standard. In particular, a temporal formula φ is valid if it is true of all behaviors. Lamport also provides a set of proof rules for TLA. In particular, he states that the proof rules reproduced in Figure 5.1, plus ordinary first-order reasoning, are sufficient (in the sense of relative completeness [Apt 1981]) for reasoning about algorithms specified in TLA without quantification over state variables. These rules should be read as asserting that if the hypotheses are valid, then so is the conclusion. For example, rule STL1 is the

$$\begin{array}{ll}
 \text{STL1. } \frac{\varphi}{\square \varphi} & \text{STL4. } \frac{\varphi \Rightarrow \psi}{\square \varphi \Rightarrow \square \psi} \\
 \text{STL2. } \square \varphi \Rightarrow \varphi & \text{STL5. } \square(\varphi \wedge \psi) \equiv \square \varphi \wedge \square \psi \\
 \text{STL3. } \square \square \varphi \equiv \square \varphi & \text{STL6. } \diamond \square(\varphi \wedge \psi) \equiv \diamond \square \varphi \wedge \diamond \square \psi \\
 \text{TLA1. } \frac{P \wedge e' = e \Rightarrow P'}{\square P \equiv P \wedge \square(P \Rightarrow P')_e} & \text{TLA2. } \frac{P \wedge [A]_e \Rightarrow Q \wedge [B]_f}{\square P \wedge \square[A]_e \Rightarrow \square Q \wedge \square[B]_f} \\
 & \text{Lattice. } \frac{\begin{array}{c} (S, \prec) \text{ is a well-founded ordering} \\ \forall x \in S: \varphi(x) \rightsquigarrow (\psi \wedge \exists y \in S: y \prec x \wedge \varphi(y)) \\ (\exists x \in S: \varphi(x)) \rightsquigarrow \psi \end{array}}{\forall x \in S: \varphi(x) \rightsquigarrow \psi}
 \end{array}$$

Figure 5.1 Proof rules for simple TLA.

well-known necessitation rule of modal logic, and it is justified because any suffix of a behavior is again a behavior, of which φ is true by the hypothesis of the rule. Of course, the implication $\varphi \Rightarrow \square \varphi$ is not valid in general. From the elementary rules of Figure 5.1, further useful verification rules can be derived, such as the following rule for proving that a state predicate P is an invariant of a system specification:

$$\text{INV1. } \frac{P \wedge [A]_e \Rightarrow P'}{P \wedge \square [A]_e \Rightarrow \square P}.$$

This rule, like most other rules for system verification in TLA, establishes a conclusion expressed in temporal logic from nontemporal (action-level) hypotheses. In this way, reasoning at the temporal level is confined to the top levels of a TLA proof and typically represents less than 5% of the proof steps. In particular, reasoning about safety properties does not involve temporal logic.

Lamport's rules are intended for the verification of algorithms. In contrast, providing a proof system for—even propositional—TLA that is complete in the standard sense of formal logic (i.e., that allows all valid formulas to be derived as theorems) is more delicate. In particular, whereas INV1 is sufficient for proving invariants of systems, the general induction axiom of LTL

$$\square(\varphi \Rightarrow \circ\varphi) \Rightarrow (\varphi \Rightarrow \square\varphi)$$

cannot be expressed in TLA because there is no next-time operator that could be applied to temporal formulas. A generalization of TLA, together with a system of rules that is complete for the propositional fragment of that logic, appears in [Merz \[1999\]](#).

5.2.3 Refinement, Hiding, and Composition

We mentioned above that TLA does not formally distinguish between system specifications and their properties: both are represented as temporal formulas. In practice, specifications of state machines are usually written in the form

$$Init \wedge \square[Next]_v \wedge L \tag{5.1}$$

where the state predicate $Init$ specifies the possible initial states of the system, the action $Next$ represents its next-state relation, v is the tuple of all state variables used in the specification, and L expresses fairness conditions. Typically, $Next$ is a disjunction of actions A_i that describe atomic transitions of the system or of its environment, and L is a conjunction of strong or weak fairness conditions on (some of) the actions A_i .

Refinement of Specifications. Perhaps influenced by ideas on program refinement developed by [Back \[1981\]](#) and [Morgan \[1990\]](#), ultimately inspired by [Dijkstra \[1976\]](#), [Lamport \[1983d\]](#) already observes that “temporal logic supports hierarchical specification and reasoning in a simple, natural way.” He also notes that the essential ingredient for this to be possible is the invariance of temporal logic formulas under stuttering, ensured by the absence of a next-time operator. The idea is that a refinement R of a high-level specification S introduces implementation detail, represented by additional state variables. Newly introduced actions that modify solely these new variables correspond to stuttering steps at the level of S and cannot invalidate S . Actions that modify the variables present in S must do so in ways that are allowed by (the next-state relation of) S . Whereas R may have fewer behaviors (when projected to the state space of S), the fairness conditions in S must be preserved: if a high-level action A_i for which S states a fairness condition is sufficiently often enabled in a run, R must ensure that some action whose effect corresponds to the occurrence of A_i will eventually occur in that run.

Summing up, R refines S if and only if the implication $R \Rightarrow S$ is valid. Assuming that R and S are specified using formulas of shape (5.1) (with superscripts R and S), refinement is proved by finding a state predicate I such that all of the following implications hold:

$$\begin{aligned} \text{Init}^R &\Rightarrow I \wedge \text{Init}^S \\ I \wedge [\text{Next}^R]_{v^R} &\Rightarrow I' \wedge [\text{Next}^S]_{v^S} \\ \square I \wedge \square [\text{Next}^R]_{v^R} \wedge L^R &\Rightarrow L^S \end{aligned}$$

In words, I is an invariant of the low-level specification R that is strong enough to prove that every transition according to R ’s next-state relation is also a possible transition for S , possibly stuttering, and to show that R implies the liveness hypotheses asserted by S . The first two proof obligations establish the safety part of the refinement and do not involve temporal logic; the third one concerns liveness and requires temporal reasoning.

Hiding of Internal State. The standard form (5.1) of specifications is useful for describing a system as a state machine, but it does not distinguish between variables that are visible at the interface and those that represent the internal state of the machine. This distinction is, however, important in the sense that the “contract” between the implementation of a system and its users should only constrain the interface, not the internal state. For example, a high-level specification S_{set} of a

key-value store may represent the current content of the store in a variable $store$ holding a set of pairs (k, v) , but the implementer should be free to choose another suitable data structure, such as a hash table. Because the operations of the lower-level specification S_{tbl} update a hash table instead of a set of pairs, the implication $S_{tbl} \Rightarrow S_{set}$ cannot be proved. Indeed, the internal variable $store$ (and the set of values that it holds) is not part of the interface of the system, which consists solely of the input and output channels through which the system corresponds with its environment. Lamport realized that this form of *information hiding* corresponds to existential quantification: the actual high-level specification of our system is not S_{set} , but rather $\exists store : S_{set}$, which asserts that the system behaves as if it contained a store represented as a set of key-value pairs. Similarly, the lower-level specification is $\exists store : S_{tbl}$,³ and in order to establish refinement, we have to prove the implication

$$(\exists store : S_{tbl}) \Rightarrow (\exists store : S_{set}). \quad (5.2)$$

Applying standard quantifier rules, that proof can be reduced to proving the implication

$$S_{tbl} \Rightarrow (\exists store : S_{set})$$

where the internal state of the lower-level specification has been exposed. It now suffices to find a suitable state function s that provides a witness term for the internal state of the higher-level specification, i.e., such that the implication

$$S_{tbl} \Rightarrow (S_{set} \text{ WITH } store \leftarrow s) \quad (5.3)$$

is provable. (The notation used here for substituting the variable $store$ by the expression s is not part of TLA.) In our example, a suitable witness s is provided by the contents of the hash table.

State functions that serve as witness terms for refinement proofs are known as *refinement mappings*. They serve to reconstruct the value of an internal state variable used in the high-level specification from the corresponding lower-level state. Semantically, showing an implication of the form (5.2) requires exhibiting an infinite sequence of values for the quantified variable on the right-hand side, given a sequence of values for the variable on the left. In contrast, a refinement mapping

3. Of course, it is immaterial if the names of the bound variables in the two specifications are the same or not.

as in (5.3) computes values state by state, which is clearly weaker: one cannot refer to previous or future values in the sequence of values satisfying the left-hand specification. Indeed, in general a suitable refinement mapping need not exist even if (5.2) holds. [Abadi and Lamport \[1991\]](#) suggested that for the proof of refinement, the low-level specification may be augmented by *auxiliary variables*. The augmented specification is semantically equivalent to the original one, but the additional variables help in defining a refinement mapping. In particular, Abadi and Lamport defined principles for augmenting specifications by *history* and *prophecy* variables, and they provided sufficient conditions for these principles to be complete for proving refinement. The constructions were presented in a semantic framework independent of TLA. Proof rules for introducing auxiliary variables in TLA appear in [Lamport and Merz \[2017\]](#).

Representing Parallel Composition. The above discussion has shown that refinement and hiding can be represented in TLA using implication and quantification, and therefore standard principles of logical deduction can be applied to reason about these concepts. Now consider two components, specified by TLA formulas Φ and Ψ , that are intended to operate in parallel. In order for both parallel components to adhere to their specifications, the variables of the first component must evolve as prescribed by Φ , and similarly for the second component. In particular, any steps that change a variable shared by both components must be permitted by both Φ and Ψ . Transitions that exclusively modify variables from one specification appear as stuttering steps to the other one and are trivially allowed by that specification, whereas variables shared between Φ and Ψ synchronize transitions of the two specifications. It follows that the formula $\Phi \wedge \Psi$ characterizes the parallel composition of the two specifications. (We see in Section 5.3.3 that for some computational models, it may be useful to adopt a stronger specification $\Phi \wedge \Psi \wedge \Xi$, where Ξ expresses extra synchronization hypotheses embodied in the computational model.)

Expressing composition as conjunction corresponds to the overall philosophy of TLA that structural concepts are expressed by logical operators. Although the conjunction of two specifications Φ and Ψ written in the standard form of (5.1) is not itself in standard form, it can easily be transformed into standard form by applying the equivalence

$$\square[A]_u \wedge \square[B]_v \equiv \square[(A)_u \wedge (B)_v]_{\langle u, v \rangle}$$

Specifying Open or Closed Systems. When specifying a component, one invariably has to describe not just the component itself but also the environment that the component is going to operate in. A closed-system specification of a component describes the most general environment that is acceptable, together with the component itself. In particular, the specification's next-state relation will be a disjunction of actions that describe the component's transitions and of actions that describe steps of the environment. Although this style works well in practice, it does not yield the most general specification of the overall system. Instead of constraining the environment, we may want to write a specification that allows the environment to behave arbitrarily, but that leaves the behavior of the component unconstrained after a step occurs that is disallowed by the assumptions on the environment. Abadi and Lamport considered ways of writing specifications that separate the environment assumptions E and the component guarantees C . The implication $E \Rightarrow C$ is a natural way for expressing such an assumption-guarantee specification, and this form is explored in [Abadi and Lamport \[1993\]](#). However, the implication holds of behaviors in which first the component violates C , and later the environment violates E . In later work, [Abadi and Lamport \[1994, 1995\]](#) introduced the stronger operator $\stackrel{+}{\Rightarrow}$ such that $E \stackrel{+}{\Rightarrow} C$ requires that (the safety part of) C may be violated only if (the safety part of) E was violated strictly earlier.⁴ Given two components described through assume-guarantee specifications, one may wish to prove that their composition refines a higher-level specification of the composed system. This is expressed in TLA as a proof obligation of the form

$$(E_1 \stackrel{+}{\Rightarrow} C_1) \wedge (E_2 \stackrel{+}{\Rightarrow} C_2) \Rightarrow (E \stackrel{+}{\Rightarrow} C)$$

In order to establish the overall system guarantee C from the component guarantees C_1 and C_2 , one will need to show that the environment assumptions E_1 and E_2 hold true. Now, the environment of each component consists of the overall environment (assumptions on which are expressed by E) and the other component, so one will want to use both E and C_2 for establishing E_1 , and similarly for the other component. Despite the apparent circularity of this reasoning chain, [Abadi and Lamport \[1995\]](#) give rules that support this approach in a sound way, based on a form of computational induction that is embodied in the definition of the $\stackrel{+}{\Rightarrow}$ operator.

4. $\stackrel{+}{\Rightarrow}$ can actually be expressed in TLA, but it is useful to consider it as a separate operator.

5.3

The Specification Language TLA⁺

Lamport designed TLA as a variant of linear-time temporal logic that is particularly appropriate for specifying executions of fair state machines. Stuttering invariance is key for representing composition as conjunction of specifications, and refinement as validity of implication. Quantification over state variables adds significant expressive power and is useful notably for distinguishing the state visible at the interface of a component from the internal state used for its implementation.

However, TLA is not a full specification language: it does not fix the interpretation of elementary function and predicate symbols such as $+$ and \in . These symbols are provided by an underlying mathematical language based on first-order or higher-order logic. In particular, nontemporal proof obligations that arise during the verification of a system property or during a refinement proof should then be discharged using a (possibly mechanized) proof system associated with that host language.

5.3.1 Overall Design of TLA⁺

Starting in the early 1990s and encouraged by successful experiments with TLP [Engberg et al. 1992], a prototype proof system for TLA based on the Larch Prover, Lamport developed the specification language TLA⁺. The language is described in the book *Specifying Systems* [Lamport 2002]; the Hyperbook and Lamport's video series [Lamport 2015, 2018] provide excellent tutorial introductions, whereas the description of TLA⁺ in Merz [2008] focuses on the semantics of the language.

TLA⁺ is based on a variant of Zermelo-Fraenkel set theory with choice (ZFC) for describing the data manipulated by an algorithm. ZFC is widely accepted by mathematicians as the basis for formalizing mathematical theories. In order to emphasize the expressiveness of ZFC, Lamport shows that a formal definition of the Riemann integral can be given in just 15 lines starting from a standard module defining the real numbers with ordinary arithmetical operators [Lamport 1992]. When writing high-level specifications of algorithms, it is useful to model data in terms of concepts such as sets and functions rather than using low-level data types provided by programming languages and their libraries. In this respect, TLA⁺ adopts a similar approach to the specification languages Z and (Event-)B [Abrial 1996, Abrial 2010, Spivey 1992]. However, the latter languages impose a typing discipline on set theory, whereas TLA⁺ is untyped. Again, Lamport follows classical mathematical practice and, for example, considers that the set $\{2, 4, 6, \dots\}$ of positive even numbers can be viewed as a type just like the set of all integers. He maintains that imposing a

decidable type system on a specification language leads to unacceptable restrictions of the expressiveness of that language. Also, embedding partial operations in a typed language often leads to objectionable choices. For example, declaring integer division as a binary operation with integer arguments and result asserts that division by 0 returns an integer, whereas implementations naturally raise an exception.⁵ Although TLA⁺ is untyped and handles partial operators by underspecification [Gries and Schneider 1995], this does not preclude tools for TLA⁺ from contracting types when this is convenient for their analyses. An article by Lampert and Paulson [1999] contains an interesting discussion of these questions.

5.3.2 A Glimpse of TLA⁺

TLA⁺ specifications are organized in *modules*. A module can extend other modules; semantically, this is equivalent to copying the content of the extended modules (with duplicates removed) into the extending module.

A module may declare constant and variable parameters. Any symbol that occurs in an expression in the module must be either a built-in symbol of TLA⁺, a parameter in the context in which the expression appears, or a symbol that was previously defined or declared.

Modules may assert properties of constant parameters in the form of assumptions or axioms, both of which express hypotheses of the module.⁶ Modules may also state theorems that can be proved using TLAPS, the TLA⁺ Proof System.

The bulk of the contents of a typical TLA⁺ module consists of definitions of operators, used to build up more complex expressions. An operator may take zero or more arguments, including operator arguments (whose arity must be specified). For example, the definition

$$\text{Symmetric}(R(_, _), S) \triangleq \forall x, y \in S : R(x, y) \equiv R(y, x)$$

introduces an operator characterizing a symmetric binary relation R over a set S . Besides the ordinary operators of first-order set theory, TLA⁺ also borrows a few constructions from programming languages, such as conditional expressions (including n -ary case distinctions) and local definitions introduced through LET-bindings.

5. Some proof assistants such as Isabelle/HOL go even further and define $n \text{ div } 0 = 0$, which is unlikely to hold in an implementation and may actually mask errors.

6. TLC will verify that an assumption evaluates to true for the concrete values substituted for the module parameters, but it will not evaluate axioms.

```

----- MODULE FIFO -----
EXTENDS Sequences
CONSTANTS Data, null
ASSUME null  $\notin$  Data
VARIABLES in, q, out
-----  

TypeOK  $\triangleq$  in  $\in$  (Data  $\cup$  {null})  $\wedge$  out  $\in$  (Data  $\cup$  {null})  $\wedge$  q  $\in$  Seq(Data)
Init  $\triangleq$  in = null  $\wedge$  out = null  $\wedge$  q = ()
Enq  $\triangleq$   $\wedge$  in'  $\in$  (Data  $\cup$  {null})  $\setminus$  {in}
 $\wedge$  q' = IF in'  $\in$  Data THEN Append(q, in') ELSE q
 $\wedge$  out' = out
Deq  $\triangleq$   $\wedge$   $\vee$  q = ()  $\wedge$  out' = null  $\wedge$  q' = q
 $\vee$  q  $\neq$  ()  $\wedge$  out' = Head(q)  $\wedge$  q' = Tail(q)
 $\wedge$  in' = in
vars  $\triangleq$  (in, q, out)
FIFO  $\triangleq$  Init  $\wedge$   $\square$ [Enq  $\vee$  Deq]vars  $\wedge$  WFvars(Deq)
-----  

THEOREM FIFOType  $\triangleq$  FIFO  $\Rightarrow$   $\square$  TypeOK
THEOREM InOut  $\triangleq$  FIFO  $\Rightarrow$   $\square$ [in' = in  $\vee$  out' = out]vars
THEOREM Liveness  $\triangleq$  FIFO  $\Rightarrow$   $\forall$  x  $\in$  Data: (in = x)  $\rightsquigarrow$  (out = x)
-----
```

Figure 5.2 TLA⁺ specification of a FIFO queue.

A module containing a system specification usually defines operators corresponding to the initial condition, the next-state relation, the overall specification, and properties to be verified. As a concrete example, Figure 5.2 contains a TLA⁺ specification of a FIFO queue. It extends the library module *Sequences*, which defines the set *Seq(S)* of finite sequences that contain elements of *S* and operations such as *Head* and *Tail* to access the first element and the remaining elements of a sequence. Module *FIFO* then declares two constants *Data* and *null* that correspond to the data to be stored in the queue and a “null” element representing the absence of data. The module also declares the variables *in*, *out*, and *q* that are used for specifying the state machine describing the behavior of the FIFO queue. Concretely, *in* and *out* represent the channels for data input and output, whereas *q* contains the current contents of the queue.

Again, TLA⁺ is untyped, and consequently one does not declare types for constant or variable parameters. Because TLA⁺ is based on set theory, there is no need to assert that *Data* is a set. In fact, semantically all values are sets, although it is

more useful to think of the elements of set $Data \cup \{null\}$, as well as of numbers or strings, as atomic values.

The second block of the module contains operator definitions.⁷ The first operator corresponds to the (intended) type invariant of the specification. The definitions of the operators *Init*, *Enq*, and *Deq* introduce the initial condition and the enqueue and dequeue actions of the queue. The initial predicate simply requires that the input and output channels contain the *null* value and that the queue is empty. The enqueue action models a change of value at the input channel. If a data value is sent over the channel, it is appended to the current contents of the queue. Otherwise (i.e., if a null value appears on the channel), the queue remains unchanged. The output value of the queue remains unchanged during an enqueue operation. Symmetrically, a dequeue operation does not modify the input channel. It puts *null* on the output channel and leaves the queue unchanged if the queue is empty and otherwise sends *Head(q)* on the output channel and removes that element from the queue. Formula *FIFO* represents the overall queue specification. Its next-state relation is the disjunction of the enqueue and dequeue actions. The fairness conjunct requires dequeue actions to happen eventually so that the queue must eventually output the values it stores.

The third block of the module states three theorems. The first theorem asserts that the type correctness predicate holds throughout any execution. The second theorem states that the values of the input and output channels never change simultaneously, and the third theorem asserts that every data value that appears on the input channel will eventually be output by the queue. We see in Section 5.5 how these properties can be verified using the TLA⁺ tools.

5.3.3 Composing Modules

Beyond module extension, TLA⁺ offers *instantiation* as a second way for composing modules. An instance conceptually creates a copy of the original module in which the constant and variable parameters can be instantiated by (constant and state) expressions. This construction is useful for composing specifications. For example, module *TwoFIFO* of Figure 5.3 declares the composition of two FIFO queues by creating two instances *Left* and *Right* of the *FIFO* module of Figure 5.2. Instance *Left* uses variables *q1* and *mid* for the internal queue and the output channel; all other parameters are instantiated by the parameters of the same name declared in module *TwoFIFO*. Similarly, instance *Right* uses *mid* and *q2* for *in* and *q*. The conjunction of the two instantiated specifications *Left!FIFO* and *Right!FIFO* describes the com-

7. The horizontal bars are decorative and have no semantic meaning.

```

MODULE TwoFIFO
EXTENDS Sequences
CONSTANTS Data, null
ASSUME null  $\notin$  Data
VARIABLES in, q1, mid, q2, out
Left  $\triangleq$  INSTANCE FIFO WITH  $q \leftarrow q1, out \leftarrow mid$ 
Right  $\triangleq$  INSTANCE FIFO WITH  $in \leftarrow mid, q \leftarrow q2$ 
Conc  $\triangleq$  INSTANCE FIFO WITH  $q \leftarrow q2 \circ q1$ 
Interleave  $\triangleq$   $in' = in \vee out' = out$ 
TwoFIFO  $\triangleq$  Left!FIFO  $\wedge$  Right!FIFO  $\wedge$   $\square$ [Interleave]in,out
THEOREM Implementation  $\triangleq$  TwoFIFO  $\Rightarrow$  Conc!FIFO

```

Figure 5.3 Composition of two FIFO queues.

position of two FIFO queues that communicate through the shared communication channel *mid*. We would like to assert that the conjunction of these specifications behaves like a FIFO whose internal queue is given by the concatenation of the internal queues of the two components. The instance *Conc* represents this “longer” FIFO queue with input channel *in* and output channel *out*, and we would therefore like to assert the theorem

$$Left!FIFO \wedge Right!FIFO \Rightarrow Conc!FIFO.$$

However, this implication is not valid: the conjunction on the left-hand side allows an enqueue action of the left FIFO queue and a dequeue action of the right FIFO queue to happen simultaneously (observe that both actions leave the shared variable *mid* unchanged). In this case, the values of the channels *in* and *out* change simultaneously, and this is not allowed by specification *Conc!FIFO*. Indeed, we wrote our specification according to an interleaving model, where enqueue and dequeue actions do not happen simultaneously. We have to explicitly enforce this interleaving assumption for the composition of the two FIFO queues, as expressed in the specification *TwoFIFO*, in order to obtain the theorem *Implementation* stated at the end of the module.

5.4

PLUSCAL: An Algorithm Language

Due to its expressiveness and high level of abstraction, TLA⁺ is a very powerful language for specifying high-level designs of concurrent algorithms and systems.

However, it may feel unfamiliar to programmers, in particular due to the syntax based on mathematical logic and to the absence of explicit control flow in the specification of systems and algorithms. Lamport designed PLUSCAL [Lamport 2009] as a language for describing algorithms that combines the look and feel of pseudocode and the precision of TLA⁺. It uses primitives that are familiar from imperative programming languages for describing the control flow of an algorithm. In contrast, the data manipulated by the algorithm is represented by TLA⁺ expressions, letting the algorithm designer benefit from the abstraction afforded by set theory without being constrained by concerns of how to concretely implement data structures.

A PLUSCAL algorithm is embedded as a comment within a TLA⁺ module and has access to all operators available at that point of the module (whether defined in extended modules or locally). The PLUSCAL translator converts the algorithm into a TLA⁺ specification that is inserted into the module. The user then states properties in terms of the TLA⁺ translation and verifies them just as for any other TLA⁺ specification, using the tools described in Section 5.5.

A PLUSCAL algorithm may declare several process templates for parallel execution, and each template can have a fixed number of instances.⁸ Variables can be declared globally, representing shared state (including the communication network of a distributed system), or locally for each process. The control flow of each process is described using standard primitives of imperative languages (sequencing, conditional statements, loops, procedure calls, etc.). In addition, the two primitives

```
either { ... }      and      with( $x \in S$ ){ ... }
or { ... }
```

are available for modeling nondeterminism. The first construct can be used to introduce a fixed number of alternatives; the second one executes a block of code for some value that is chosen nondeterministically from the set S . Synchronization among processes is modeled using the instruction **await** P that blocks until the predicate P becomes true.

An important aspect for the specification of concurrent algorithms is to identify the “grain of atomicity,” i.e., which blocks of statements should be executed without interference from other processes. Rather than imposing an arbitrary fixed level of atomicity, PLUSCAL uses labels to identify yield points at which processes may be interrupted. A group of statements between two labels is assumed to be executed atomically. This allows the designer to choose the degree of atomicity appropriate for the specification and to compare algorithms described at different degrees

8. PLUSCAL does not support dynamic spawning of processes.

```

MODULE ProducerConsumer
EXTENDS Naturals, Sequences
CONSTANTS Data, maxCapacity
ASSUME maxCapacity ∈ Nat \ {0}
(*
-algorithm ProducerConsumer {
    variable q = ⟨⟩;
    define {
        nonempty  $\triangleq$  Len(q) > 0
        nonfull  $\triangleq$  Len(q) < maxCapacity
    }
    process (Producer = “p”) {
        p: while (TRUE) {
            await nonfull;
            with (item ∈ Data) {
                q := Append(q, item)
            } } }
        fair process (Consumer = “c”)
            variable rcvd;
            c: while (TRUE) {
                await nonempty;
                rcvd := Head(q);
                q := Tail(q)
            } }
    }
*)

```

Figure 5.4 A specification of a producer-consumer system in PLUSCAL.

of atomicity.⁹ In order to ensure liveness of PLUSCAL algorithms, fairness conditions may be attached to labels or to entire processes. These ensure that the group of statements following the label (respectively, the entire process) will eventually execute if it is enabled sufficiently often.

As an example, a PLUSCAL algorithm modeling a simple producer-consumer system appears in Figure 5.4. It declares two process templates for the producer and

9. Some rules govern where labels must or cannot be placed, essentially to ensure that PLUSCAL algorithms are easy to translate into TLA⁺ specifications.

the consumer, each of which is instantiated once for process identities “*p*” and “*c*.” The two processes communicate using a shared FIFO queue of bounded capacity *maxCapacity*. Each process has an infinite loop: the producer repeatedly adds new data to the queue, while the consumer retrieves the data from the queue. By declaring a (weak) fairness condition for the consumer process, we ensure that every data item that is present in the queue will eventually be consumed. In this specification of the algorithm, the bodies of the while loops execute atomically; nonatomic execution would be modeled by inserting additional labels. The operations *Len*, *Append*, *Head*, and *Tail* that appear in the presentation of the algorithm are defined in the standard module *Sequences* that is extended by module *ProducerConsumer*.

Invoking the PLUSCAL translator on module *ProducerConsumer* generates a TLA⁺ specification corresponding to the algorithm. In particular, the translator generates declarations of TLA⁺ variables corresponding to the global and local variables of the PLUSCAL algorithm, and it derives the initial condition from the initializations of the PLUSCAL variables.¹⁰ The essential step of the translation is to generate a TLA⁺ action for each group of statements between two consecutive labels. For example, the single group of statements contained in the producer process of Figure 5.4 is represented by the action

$$\begin{aligned} \text{Producer} \triangleq & \wedge \text{nonfull} \\ & \wedge \exists \text{ item} \in \text{Data}: q' = \text{Append}(q, \text{item}) \\ & \wedge \text{rcvd}' = \text{rcvd} \end{aligned}$$

For more complicated algorithms, the translator adds a variable *pc* that represents the current point of control of each process. When a process type has several instances, their local variables are represented using arrays (i.e., TLA⁺ functions).

Because the translation from PLUSCAL to TLA⁺ is fairly direct, the generated TLA⁺ specification is usually quite readable. This is important because correctness properties of the algorithm are written in TLA⁺ rather than in PLUSCAL. For our producer-consumer example, we may want to verify the invariant *BoundedQueue* and the temporal property *Liveness*, defined as

$$\begin{aligned} \text{BoundedQueue} \triangleq & q \in \text{Seq}(\text{Data}) \wedge \text{Len}(q) \leq \text{maxCapacity} \\ \text{Liveness} \triangleq & \forall d \in \text{Data}: (\exists i \in 1.. \text{Len}(q): q[i] = d) \rightsquigarrow \text{rcvd} = d \end{aligned}$$

that express type correctness and eventual reception of every data item contained in the queue.

10. For PLUSCAL variables that are not initialized, such as *rcvd* in our example, the translator adds a default initialization, which is necessary for model checking using TLC.

5.5 Tool Support

5.5.1 The Model Checker TLC

Lamport originally designed TLA⁺ as a precise and expressive language for specifying algorithms and for (deductively) reasoning about their properties. It was used in the second half of the 1990s by hardware designers at Digital Equipment Corporation, in particular for describing cache coherence protocols of multiprocessors [Joshi et al. 2003]. They wrote rigorous, informal proofs for key invariants maintained by these protocols. Yuan Yu then recognized that it was possible to support this type of reasoning using model checking. Lamport reports that originally he was very skeptical of this idea. Input languages for model checkers such as Spin [Holzmann 2003], SMV [McMillan 1993], or Murphi [Dill et al. 1992] are based on low-level primitives carefully chosen to support efficient verification of finite-state systems, whereas TLA⁺ uses the full power of ZF set theory and is intended for modeling systems of arbitrary size. It is not possible in general to systematically enumerate all behaviors that satisfy a given TLA⁺ formula.

TLC, the TLA⁺ model checker, accepts a subset of TLA⁺ specifications written in standard form (5.1). It is an explicit-state model checker, intended for verifying finite instances of specifications. In addition to the specification, the user has to provide the model checker with a *model* that describes a finite instance by fixing specific values for constant parameters. For the queue specification of Figure 5.2, one could, for example, fix parameter values $Data = \{1, 2, 3\}$ and $null = 0$. TLC then interprets the specification, restricted to this model, by decomposing the next-state relation into disjuncts (bounded existential quantification over finite sets is expanded into an explicit disjunction) and evaluating each disjunct from left to right. The first occurrence of a primed variable v' has to be of the form $v' = e$ or $v' \in e$ for an expression e that TLC can evaluate; in the second form, e must evaluate to a finite set. The first form is interpreted as an assignment of (the value denoted by) e to v in the successor state. The second form leads to the generation of one successor state per element of e , with v assigned to that element. Subsequent occurrences of v' are interpreted by the value assigned to v in the successor state in this way. For example, the conjunct

$$in' \in (Data \cup \{null\}) \setminus \{in\}$$

of the action Enq of Figure 5.2 generates one successor state for each element of $Data \cup \{null\}$, except for the current value of in . The occurrences of in' in the second conjunct of Enq then refer to the value chosen for that successor state. The initial predicate is evaluated in a similar way. TLC aborts with an error message if the initial

predicate or some subaction of the next-state relation does not assign a value to some of the variables declared in the module.

When evaluating set-theoretic expressions, TLC will generally enumerate the elements, but it will apply some optimizations. For example, in evaluating the predicate $e \in \text{Nat}$ that may occur in a typing invariant, TLC simply checks if (the value denoted by) e is a natural number. TLC disallows unbounded quantification, and it will signal an error when it would have to enumerate an infinite set.

Using the strategy outlined above, TLC enumerates all reachable states in a breadth-first manner, and it checks the invariant predicates provided by the user during this state enumeration. When an invariant evaluates to false, the run is aborted and a counterexample is displayed. (Due to breadth-first search, that counterexample will be of minimal length.) Liveness properties are evaluated over the state graph computed during state enumeration, based on the tableau algorithm of [Lichtenstein and Pnueli \[1985\]](#). TLC parallelizes state enumeration on multicore machines and provides a distributed implementation for running in a cluster or cloud environment. States may be stored to disk so that state exploration is not memory bound, and TLC regularly performs checkpoints so that model checking can be resumed in case of a crash. In order to limit the explored state space, the user can impose state constraints. For example, the FIFO queue of Figure 5.2 generates an unbounded state space even for a fixed finite set $Data$ because the length of the queue can grow without bound, and the user can choose not to explore successors of states in which $\text{Len}(q)$ exceeds some fixed value. TLC also implements symmetry reduction in order to explore only a quotient of the state space with respect to an equivalence relation. In the queue example, the user may choose to declare $Data$ (or more precisely, the set that the parameter $Data$ is instantiated with in the concrete model) as a symmetry set because all operations are insensitive to particular values in that set.

Although TLC imposes certain restrictions on the specifications that it can check, most specifications that are written in practice adhere to those restrictions or can easily be rewritten so that they do. (The fact that TLC has been the main analysis tool for TLA⁺ specifications has also contributed to disciplining users so that they respect those restrictions.) In particular, TLA⁺ specifications obtained from translating PLUSCAL algorithms can be checked using TLC. The different properties asserted in the modules of the previous sections can be verified by TLC for concrete instances of parameters, including the theorems of Figures 5.2 and 5.3, as well as the properties of the producer-consumer system given at the end of Section 5.4.¹¹

11. The specification *TwoFIFO* of Figure 5.3 needs to be rewritten in standard form so that TLC can verify it.

Like most model checkers, TLC is most useful when a counterexample to a putative property is discovered. A positive verdict only means that the checked properties hold for the particular model that TLC checked, and it requires sound engineering judgment to determine if this gives enough confidence in the correctness of the properties for arbitrary instances of the specification.

5.5.2 The TLA⁺ Proof System

TLAPS, the TLA⁺ Proof System [Cousineau et al. 2012], is a proof assistant for checking proofs written in TLA⁺. For this purpose, TLA⁺ was extended to include a language for writing hierarchical proofs based on a format that Lamport had proposed earlier [Lamport 1995] for writing rigorous pencil-and-paper proofs. For example, the proof of type correctness for the FIFO queue of Figure 5.2 can be written as follows.

```

THEOREM FIFOType  $\triangleq$  FIFO  $\Rightarrow$   $\square$  TypeOK
(1)1. Init  $\Rightarrow$  TypeOK
    BY DEF Init, TypeOK
(1)2. TypeOK  $\wedge$  [Enq  $\vee$  Deq]vars  $\Rightarrow$  TypeOK'
    BY DEF Enq, Deq, vars, TypeOK
(1)3. QED
    BY (1)1, (1)2, PTL DEF FIFO

```

Following a standard pattern for invariance proofs (cf. rule INV1 of Section 5.2.2), the first two steps of the proof establish that the initial predicate of the *FIFO* specification implies predicate *TypeOK*, and that the predicate is preserved by every step allowed by the next-state relation. The third step concludes the proof of the theorem. The justifications for each step are indicated following the keyword BY. For the first two steps, it suffices to expand the relevant definitions and then apply built-in automatic proof back-ends that mechanize standard mathematical reasoning. The justification of the third step uses the assertions of the two preceding steps and propositional temporal logic; it also expands the definition of *FIFO* in order to expose its initial and next-state predicates. Because the proof is so simple, we only need one level of proof: all step names have the form $\langle 1 \rangle n$. The steps of more complicated proofs can be decomposed into lower-level proof steps until TLAPS can prove the leaf steps of the proof automatically.

Figure 5.5 schematizes the architecture of TLAPS. The central component is the proof manager that interprets the proof language, maintains the context of

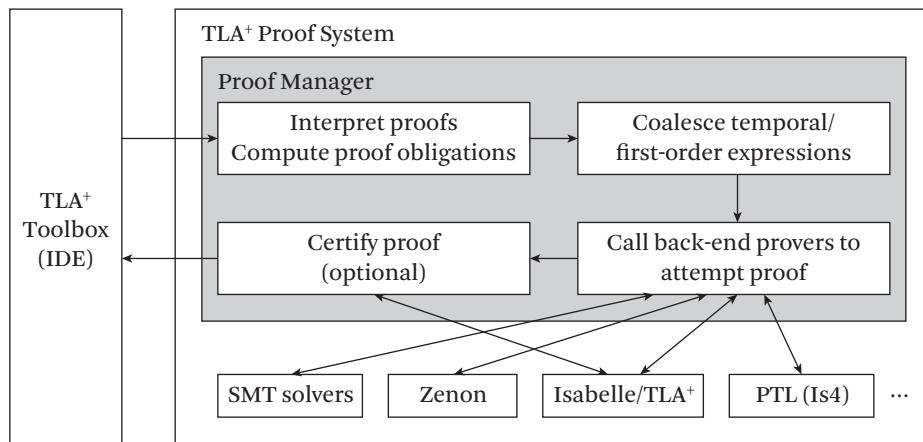


Figure 5.5 Architecture of the TLA⁺ Proof System.

each proof step (i.e., the visible identifiers, assumptions, and definitions) and computes the corresponding proof obligations. In a nontemporal step (such as the first two steps in the above example), primes are pushed inside complex expressions, and then primed symbols are replaced by fresh identifiers. Similarly, any temporal expressions appearing in the context are abstracted by fresh predicate symbols. Similarly, in a temporal step (such as the QED step above), any first-order formulas are abstracted by propositional variables. This transformation is called *coalescing* [Doligez et al. 2014]; it is necessary so that back-end provers see the proof obligation either as a standard formula of mathematical set theory or as a propositional temporal logic formula. The proof manager then calls back-end provers to attempt and prove the proof obligation. Currently, TLAPS supports SMT solvers (via a translation to the SMT-LIB2 language [Barrett and Tinelli 2018]), the tableau prover Zenon [Bonichon et al. 2007], and an encoding of TLA⁺'s mathematical set theory as an object logic in the logical framework Isabelle [Paulson 1994] for proving nontemporal steps. It also comes with a decision procedure for propositional temporal logic [Suda and Weidenbach 2012] for proving temporal proof obligations. The architecture is open for supporting additional back-end provers through suitable translations of proof obligations into their input language. For increased confidence in the correctness of TLAPS proofs, when a back-end prover finds a proof it may return a justification to the proof manager for checking by the trusted kernel of Isabelle. This certification step is optional and currently only available for proofs found by Zenon.

TLAPS is currently restricted to proving safety properties. The planned extension to liveness properties requires support for handling ENABLED predicates and for first-order temporal logic reasoning, for example, for mechanizing the Lattice rule of Section 5.2.2. TLAPS has been used for verifying several distributed algorithms, including variants of Paxos [Lamport 2011, Chand et al. 2016] and a version of the Pastry distributed hash table [Azmy et al. 2018].

5.5.3 The TLA⁺ Toolbox

Editing and analyzing TLA⁺ specifications is facilitated by the TLA⁺ Toolbox, an Eclipse application that provides an IDE (integrated development environment) for TLA⁺. It provides support for editing TLA⁺ specifications and proofs, such as looking up operator definitions, properly indenting TLA⁺ specifications, renumbering proof steps, and hiding subproofs that are irrelevant for the current branch. The Toolbox is integrated with the TLA⁺ tools, including SANY, the TLA⁺ syntactic and semantic analyzer, the TLATEX pretty printer, TLC, and TLAPS. In particular, the user interface to TLC provided by the Toolbox greatly simplifies the definition of finite-state models to be verified, the analysis of counterexamples, and the evaluation of TLA⁺ expressions.

All TLA⁺ tools are released as open-source software under licenses for use in industry or academia.

5.6 Impact

The concepts that Lamport introduced for the formal specification and verification of algorithms have deeply influenced the research community. The notions of safety, liveness, and fairness are universally recognized for their fundamental importance. The concept of stuttering invariance is valuable in contexts other than those strictly related to refinement and composition; in particular, it plays an important role in partial-order reductions used for model checking distributed systems [Godefroid and Wolper 1994, Valmari 1990]. The idea of writing system specifications in terms of state machines is widely accepted [Abrial 2010, Gurevich 1995]. The specification language TLA⁺ is taught at universities around the world, and PLUSCAL is starting to be used as a vehicle for teaching courses on distributed algorithms.

The first significant use of TLA⁺ in industry was for specifying and verifying cache coherence protocols by the group of hardware engineers that designed Digital Equipment Corporation's Alpha processors [Joshi et al. 2003]. Members of that group subsequently moved to Intel and continued to use TLA⁺, although little is

publicly known about the impact of that work. Work at Microsoft using TLA⁺ started around 2003 with the specification of the Web Services Atomic Transaction protocol [Johnson et al. 2007]. This experience was considered successful, and engineers at Microsoft continued to use TLA⁺. Reportedly, use of TLA⁺ contributed to identifying a serious error in the XBox 360 memory system that would have been difficult to debug using conventional techniques. The Farsite project [Bolosky et al. 2007] at Microsoft Research developed a scalable, serverless, and location-transparent distributed file system that could tolerate nodes being unavailable, as well as malicious participants. The designers used TLA⁺ for specifying the distributed directory service and refined a centralized functional specification into the formal description of a distributed protocol. They report that the main benefit of using formal specification and verification was to understand the invariants that the system must maintain through different levels of refinement. They consider that it would have been far more costly to iterate through several designs at the implementation level where aspects related to the distributed protocol would have been mixed with low-level coding details. In contrast, they found that developing an implementation from the protocol specification was rather straightforward because only sequential code had to be written, without a need for thinking about aspects related to distributed execution. In the later IronFleet project [Hawblitzel et al. 2015, 2017], researchers at Microsoft pushed this idea even further. Combining a TLA-style approach to state machine specification and refinement with a Floyd-Hoare style of reasoning about imperative programs provided by Dafny [Leino 2010], they obtained a mechanized framework for designing, implementing, and verifying distributed systems from high-level (centralized) specifications to distributed protocols and further to executable code that exhibited competitive performance. Based on an embedding of TLA and its proof rules in Dafny, they could prove not only safety but even liveness properties in a unified framework. The approach was used to develop a replicated state machine library and a sharded key-value store.

An interesting account of the use of TLA⁺ in industry was provided by a group around Chris Newcombe working at Amazon Web Services [Newcombe et al. 2015]. They reported that not only have TLA⁺ specifications contributed to finding subtle bugs in high-level designs of distributed protocols, but the understanding and confidence obtained from formal specification and verification allowed them to make aggressive performance optimizations without sacrificing correctness. Several other companies developing web and cloud services, including the groups working on Azure at Microsoft, actively use TLA⁺ and TLC for describing and ver-

ifying the protocols they design. The TLA⁺ Google group¹² and regular in-person community meetings provide forums for the members of the TLA⁺ community to exchange and help each other in case of problems.

TLA⁺ is intended as a formalism for modeling and verifying high-level designs of algorithms and systems. Doing so does not prevent coding errors from creeping into implementations of verified algorithms: such errors can be caught using techniques of program verification. However, the implementation of a buggy design is virtually guaranteed to contain the design errors, and finding and fixing these issues at the level of executable code is much more difficult and costly than doing so at an early stage of development, using specifications written at the appropriate level of abstraction.

12. <https://groups.google.com/forum/#!forum/tlaplus>

Biography

Roy Levin

6.1

Early Years

Leslie Lamport was born in 1941, the year the United States entered World War II. Computers were specialized and primitive. Computing, as a discipline, didn't exist. Vannevar Bush's famous essay *As We May Think* [Bush 1945], which foresaw the modern computing world of interconnected computers on which Lamport would have enormous impact, wouldn't appear for another four years.

Lamport grew up in New York City. His interest in mathematics began in elementary school, and he recalls that his mother taught him long division years before he heard about it in the classroom, which would normally have been around age 9. Learning arithmetic at home was perhaps more common in the years before computers became commonplace, as the focus of grade-school mathematics education was to teach "shopkeeper math," which most parents had learned themselves in the same way. Now, of course, many parents can't do division without a calculator.

The atmosphere of Lamport's childhood home wasn't math- or science-oriented, though his father had prepared for medical school as a college undergraduate, premed being a three-year, nondegree program then. Lamport's father never went to medical school (it was the Great Depression), but he had a scientific turn of mind, and colleagues at his workplace called him "Doc" [Lamport 2018b]. Lamport recalls taking long walks with his father, during which his father told him things that stimulated an interest in physics. (Recalling those walks, Lamport wondered "if that's why I've always had the habit of getting up and walking (or pacing) when I was thinking" [ibid.].)

In 1954 at age 13, Lamport began the tenth grade at the famed Bronx High School of Science, having skipped two grades previously. He doesn't recall any interest in computers before entering high school, even though they had acquired broad

public awareness through television in 1952 when a UNIVAC computer predicted a landslide win for Dwight Eisenhower in the US presidential election.¹ Lamport's family, however, didn't acquire a television until 1954 (to see the Army-McCarthy hearings). But he did become aware of computers—more precisely, the mathematics behind the circuits out of which they were built—from a book on Boolean algebra, probably in his junior year. He also thinks he probably saw them in the movies [ibid.].

During high school, Lamport also tried to build a computer with a friend. He visited IBM in New York City and acquired some functional used vacuum tubes. (Maintenance procedures of the day dictated that tubes be replaced on a schedule rather than waiting for them to fail.) Lamport and his friend got as far as building a working four-bit counter: quite a way from a complete computer but enough to impress interviewers at MIT when Lamport later applied for admission there [Lamport and Levin 2016a, page 3].

These first explorations of computing, however, didn't rival Lamport's earlier interest in mathematics.² It was in his senior year of high school that he first entertained the idea of a career as a mathematician, though he freely confesses that he probably had no idea what that entailed [ibid.]. He showed the first signs of mathematical creativity in that year, when he published his first paper, "Braid Theory," in the *Mathematics Bulletin of the Bronx High School of Science* [Lamport 1957]. Today, he downplays its significance, saying it "shows I was not a prodigy" [Lamport 2019, comment 1]. But it shows that, even as a 16-year-old, Lamport wanted to communicate his ideas effectively. The same publication contains the companion article "An Introduction to Group Theory" by his classmate Alison Lord, which Lamport's paper references in its opening sentence as "essential" background. Evidently, the teacher responsible for the publication saw sufficient merit in Lamport's work to provide readers with the context needed to understand it.

1. This was the first use of computers for election prediction, and the CBS television network executives suppressed the prediction, believing it to be wrong since it differed sharply from conventional polling. When it proved accurate, CBS was forced to eat crow [Wikipedia 2018d].

2. Recalling his grade-school years more than a half century later, Lamport opined: "I suspect that I was attracted to math because of its simple certainty, which was in stark contrast to the unpredictable behavior of human beings. I see no point in trying to psychoanalyze that" [Lamport 2018b].

6.2

Education and Early Employment

After graduating from high school, Lamport worked for a summer at Consolidated Edison (Con Ed), the electric utility company in New York City. He began doing what he characterized as “very boring stuff” [Lamport and Levin 2016a, page 3] but managed to get transferred to the computer center where there was an IBM 705, a relatively sophisticated computer for 1957.³ Though Lamport’s job was to be a classic “glass house” computer operator—mounting/dismounting tapes and running card-deck jobs—he and the computer had enough spare time that he learned to program it. Of course, he used assembly language, as FORTRAN was just coming into existence [Backus et al. 1957]. Lamport recalls that his first program computed e to around 125 digits, that number being chosen because it matched the size of the 705’s accumulator, thereby sparing him the need to do multiple-precision arithmetic [Lamport and Levin 2016a, page 3].

After his first summer at Con Ed, Lamport began his undergraduate years at MIT. He initially planned to major in physics but switched to mathematics when he discovered that it was the only major that didn’t require an undergraduate thesis [ibid.]. Nevertheless, he continued to study physics as well, and supported himself by writing programs: during the academic year for a professor in the business school and during the summer at Con Ed. The summer job offered him a different kind of intellectual stimulation than math or physics. As Lamport recalled it [Lamport and Levin 2016a, page 11, edited]:

I had a manager who became sort of a mentor to me, and he would give me these problems. In those days, programs were on punch cards, and you would load the punch cards into the hopper, and you’d press a button on the console. The computer would read the first card and then put that card in memory and execute that piece of memory as a program. So, the first card would usually be something that had just enough of a program in it to load a few more cards, so you’d have a decent-sized program that you would then execute to load your program and start that program executing. I remember he posed the problem to me to write a loader that worked all on a single card. And I worked on that problem and I would present him the solution and he’d say, “Yeah, that’s good. Now make your loader also have this other property,” and we went through a few iterations like that. I just loved those puzzles.

3. Readers unfamiliar with computers of the era will find the technical marketing document educational [IBM 1955]. In particular, the main memory capacity (magnetic cores) was trumpeted as 20,000 *characters*.

Lamport's later focus on concurrency began with the same love of puzzles, though more than 15 years would elapse before he found the opportunity to apply his knowledge of physics to those problems. During his undergraduate years, he pursued interests in math, physics, and computing, but they were "completely separate worlds" [Lamport and Levin 2016a, page 4] to him at the time. He certainly didn't think of computing as a career, for as a math major, "the only thing I knew that mathematicians did was teach math, so I suppose my career goal was to be a professor of mathematics" [ibid.].

He undertook his graduate education in mathematics, at Brandeis University, with that objective, earning his M.S. in 1963 and his Ph.D. in 1972. During those years, computing continued to be a part-time livelihood. He worked as a programmer at MITRE Corporation (1962–65), a government contractor, then beginning in 1970 at Massachusetts Computer Associates (COMPASS), a software contracting company, on the recommendation of a Brandeis faculty member. In between, he taught undergraduate mathematics at Marlboro College (1965–69) and authored an unpublished calculus text.

As he approached his Ph.D. dissertation, Lamport initially planned to work in mathematical physics, combining his two undergraduate foci, but he instead settled on a topic in analytic partial differential equations. He characterizes his thesis as "a small, solid piece of very classical math" and adds, "I learned nothing about analytic partial differential equations except what was needed for my thesis research, and I have never looked at them since then" [Lamport 2019, comment 7].

6.3

The COMPASS Years (1970–1977)

When Lamport finished his Ph.D. in 1972, he planned to leave Boston after 15 years of education and become a professor at the University of Colorado in Colorado Springs. But he had been working part-time at COMPASS for two years and they offered him a staff position . . . in California. They had no office there, but anticipated setting one up and said that he could work for them until the office was in place. So, he turned down the academic job in Colorado and began working full-time in the computer industry, where he remained throughout his career.

COMPASS's main business was building FORTRAN compilers, something that nearly every computer sold in the 1970s needed to have. Scientific computation, for which FORTRAN was intended, often involves matrices whose elements can be operated upon in parallel if the computer has the hardware to do so and if the software can take advantage of it. But FORTRAN was conceived before parallel computation was a reality, so FORTRAN programs on matrices were customarily

written as loops, often nested, obscuring the inherent parallelism.⁴ In the early 1970s, the University of Illinois tried to build an array processing computer, the ILLIAC IV, for which COMPASS had contracted to create a FORTRAN compiler that rediscovered the parallelism in these loops.⁵ This compiler was terra incognita for COMPASS, and they turned to Lamport for help.

Lamport proposed that he work on the problem while spending a month or two with a friend in New Mexico. COMPASS agreed. Lamport worked out the theory (based on linear algebra), which he characterized as “pretty straightforward,” and created the algorithms based on the theory [Lamport 2019, comment 9]. To communicate both clearly to his colleagues, he wrote a substantial document. Such a thing was unheard of in the software industry at the time, but it had the desired effect. Lamport said, “I learned later from observation that this tome . . . was practically a sacred text that people studied, and they did use it to build the compiler” [Lamport and Levin 2016a, page 6].

In retrospect, this episode represents a significant first in Lamport’s career, for several reasons. It was the first setting in which he created a software specification with a firm mathematical foundation. It was his first substantive piece of work to appear in the prestigious *Communications of the ACM (CACM)* [Lamport 1974b], other than a short note earlier commenting on another’s work [Lamport 1970]. Perhaps most significantly, it established his ability to operate as an independent researcher in a corporate context, for on the strength of the parallelizing compiler work, COMPASS agreed to send him to California before their office existed. As things transpired, the office never materialized, but Lamport moved to the San Francisco Bay Area and worked independently for COMPASS, supported by various government contracts while carrying out self-directed research that sometimes had little immediate relevance to those contracts. As Lamport put it, the parallelizing compiler work “reassured the people at COMPASS that I could go off by myself without supervision and actually do something useful” [Lamport and Levin 2016a, page 6]. He continued to do so, visiting the COMPASS office in Massachusetts for about a month annually, for another five years. During those years he created two

4. FORTRAN was already well established as the de facto standard language for scientific computation, despite its unsuitability for expressing matrix computations. Better alternatives existed, such as Iverson’s APL [Iverson 1962], a language in which arrays are the only data structure. APL was commercially available on IBM S/360 computers in 1966. However, it was implemented as a time-shared, multiuser interpreter, making it unsuitable for many workloads. Despite its power and elegance, it never achieved widespread use.

5. ILLIAC IV’s design had four CPUs and 256 FPUs, but only one “quadrant” of the machine was built, effectively creating a single-processor machine with 64 arithmetic units [Wikipedia 2019c].

of his seminal works—the Bakery algorithm and the Time/Clocks paper—though their significance would not be recognized for many years.

Lamport's algorithms for the parallelizing FORTRAN compiler used a specialized kind of concurrent computation matched to the architecture of ILLIAC IV: a single thread of control that operates synchronously in parallel on arrays distributed across multiple processing units.⁶ By contrast, the Bakery algorithm and essentially all of Lamport's subsequent work in concurrency use multiple communicating processes, the kind of parallelism that has become ubiquitous in modern computing. Lamport described how the algorithm came about [[Lamport and Levin 2016a](#), pages 7–8, edited]:

In 1972, I became a member of the ACM. One of the first *CACM* issues that I received contained a mutual exclusion algorithm.⁷ I looked at that, and it seemed to me that that was awfully complicated: there should be a simpler solution. I decided, “Oh. Here’s a very simple solution for two processors.” I wrote it up, I sent it to ACM, and the editor sent it back saying, “Here is the bug in your algorithm.” That taught me something! It taught me that concurrency was a difficult problem and that it was essential to have proofs of correctness of any algorithm that I wrote. Well, of course, it got me mad at myself for being such an idiot. I determined to solve the problem, and in attempting to solve it, I came up with the Bakery algorithm.

Mutual exclusion of parallel processes accessing shared data is a central problem in concurrent computation. It was not new when Lamport conceived the Bakery algorithm, having received considerable attention in publications going back at least seven years earlier. Chapter 1 defines the problem and explains how Lamport's Bakery algorithm addresses it. Lamport didn't mention either bakeries or mutual exclusion in the title of his paper: “A New Solution of Dijkstra’s Concurrent Programming Problem” [[Lamport 1974a](#)]. Indeed, the analogy with bakeries doesn't appear until nearly halfway through the concise paper. As noted in [Lamport and Levin \[2016a\]](#), page 7], though the paper presents a concrete algorithm, it couldn't

6. Lamport had earlier used another specialized form of concurrency when he designed a file system for a computer being built by Foxboro Computers (again, COMPASS had the software contract for the machine). This system used interrupts, a low-level form of concurrency requiring careful discipline to program correctly [[Lamport and Levin 2016a](#), page 5]. Much of the early research in concurrency sought to create better-behaved mechanisms to replace interrupts.

7. In the early decades of computing, the *Communications of the ACM* had a monthly algorithms section.

appear in *CACM*'s algorithms section, since publication there required working code and, at the time, there were no working general multiprocessors! Despite its brevity, the paper contains several notable features beyond its technical innovation. It prominently includes a proof of the algorithm's correctness—a direct consequence of Lamport's experience quoted above. It points the way to a major area of Lamport's future work, fault tolerance, with a single concluding sentence: "Since [the algorithm] does not depend upon any form of central control, it is less sensitive to component failure than previous solutions." It devotes three paragraphs to characterizing a practical problem in implementing the algorithm: the potentially unbounded values of counters and the implications for storing them in fixed-length registers. (Lamport subsequently worked on this problem as well; see the discussion in Chapter 1.) The paper also contains the first appearance of some techniques that Lamport would employ regularly in subsequent work, such as the use of a counter as a timestamp. In fact, this kind of timestamp plays an essential role in the other seminal work of Lamport's COMPASS years, the Time/Clocks paper.

The Bakery algorithm paper of 1974 documents Lamport's first significant research result—one in which he takes pride [Lamport and Levin 2016a, page 11, edited]:

I think in other things that I've done, I can look back and see: "This idea developed from something else." Sometimes it would lead back to a previous idea of mine, very often it would lead to something somebody else had done. But the Bakery algorithm just seemed to come out of thin air to me. There was nothing like it that preceded it, so perhaps that's why I'm proudest of it.

"Time, Clocks and the Ordering of Events in a Distributed System" [Lamport 1978b]—generally shortened to the Time/Clocks paper—grew out of a working paper by Paul Johnson and Robert Thomas entitled "The Maintenance of Duplicate Databases" [Johnson and Thomas 1975], published as an RFC in 1975.⁸ Lamport received a copy of the paper, probably in 1975, and realized that the algorithm it proposed for keeping replicated databases in synchrony wasn't quite right because "it permitted a system to do things that, in ways, seemed to violate causality" [Lamport and Levin 2016a, page 20]. Lamport believes that his physics background enabled

8. The acronym RFC (Request for Comments) in this context refers to the mechanism of technical communication created as part of the ARPANET project, one of the inspirations for the Internet. RFCs, which were hand-produced in days before computer-based word processing, ran the gamut from working papers to specifications of standards.

him both to recognize the deficiency and to correct it; specifically, he could directly apply special relativity and the space-time view of Minkowski's seminal 1908 paper [Minkowski 2017]. "I realized that the problems in distributed systems are very much analogous to what's going on in relativity because in relativity there's no notion of a total ordering of events: to different observers, events will appear to happen in different orders. But there is a notion of causality, and I realized that there's an obvious analog of that in distributed systems" [Lamport and Levin 2016a, page 20, edited].

To order the events in their replicated databases paper, Johnson and Thomas used timestamps, and Lamport's solution does as well. Lamport has frequently pointed this out when people mistakenly credit him with inventing the use of timestamps for this purpose. Perhaps the fame that this paper enjoyed for its real innovations led readers to believe that everything in it was novel. Since Lamport's paper was not an explicit response to Johnson and Thomas—their paper served as the initial stimulus for Lamport's thinking—many readers were doubtless unaware of the earlier work, which did not receive the wide circulation of Lamport's *CACM* paper. Whatever the reason, it is ironic that many people consider the causal ordering of events via timestamps, and the algorithm that Lamport chose to implement it, as the key innovations of the paper. In Lamport's words [Lamport and Levin 2016a, page 21, edited]:

I realized that this algorithm was applicable not just to distributed databases, but to anything, and the way to express "anything" is a state machine. What I introduced in this paper was the notion of describing a system by a state machine. . . . What I said in the paper was: what any system is supposed to do can be described as a state machine, and you can implement any state machine with this algorithm, so you can solve any problem. The simplest example I could think of was a distributed mutual exclusion algorithm, and since distributed computing was a very new idea, this was the first distributed mutual exclusion algorithm, but I never took that seriously as an algorithm. Nor do I take the Time/Clocks paper's algorithm as the solution to all problems, because although in principle you can solve any problem this way, it didn't deal with failures. It's not clear that the solution would be efficient, and there could be better solutions. In fact, I never thought that my distributed mutual exclusion algorithm would in any sense be an efficient way of doing mutual exclusion. Well, the result of publishing the paper was that some people thought that it was about the partial ordering of the events, some people thought it was about distributed mutual exclusion, and almost nobody thought it was about state machines! And as a matter of fact, on two separate occasions, when I was discussing that paper with somebody and I said, "the really important thing is state machines," they said to

me, “There’s nothing about state machines in that paper.” I had to go back and look at the paper to convince myself that I wasn’t going crazy and that I really did mention state machines in that paper!⁹

Chapter 2 includes an overview of the Time/Clocks paper. Whether or not the field remembers the paper’s key insight about the generality of state machines for building distributed systems, the state machine approach caught on. A few months after the paper appeared, Fred Schneider, then a first-year assistant professor at Cornell University, mailed Lamport a draft of a paper [Schneider 1982] that built on the Time/Clocks paper. Schneider’s paper considered failures, which Lamport’s had not, and that first interaction led to a highly productive multidecade collaboration and a half-dozen papers exploring how to reason about concurrent programs. Inspired by the Time/Clocks paper, Schneider later produced a tutorial on state machines that became the essential reference on the state machine approach [Schneider 1990].

A great deal of Lamport’s subsequent work in concurrency can be traced back to these two seminal papers of his COMPASS years that address fundamental issues in distributed systems: mutual exclusion (Bakery algorithm) and causality (Time/Clocks). Remarkably, at that time distributed systems played a relatively small and isolated role in the computing universe. Why, then, did Lamport work on them? He said: “I don’t want to give the impression that I was drawn to concurrency by fundamental problems. The fact of the matter is that they were just really cool puzzles. Concurrency added a whole new dimension of complexity, and so even the simplest problem became complicated” [Lamport and Levin 2016a, pages 11–12, edited].

But the computing universe was rapidly changing. By the time the Time/Clocks paper appeared, Lamport had moved to a new employer, and he quickly became engaged in a project with a very real distributed system.

6.4

The SRI Years (1977–1985)

In 1977, COMPASS decided not to continue Lamport’s remote working arrangement. Lamport wanted to remain in California, so he interviewed with two well-known research labs: Xerox PARC in Palo Alto and SRI International in neighboring Menlo Park.

9. As Butler Lampson put it: “The paper is quite famous, but hardly anyone (including myself, until Leslie pointed it out to me) has noticed that it introduced replicated state machines; everyone has focused on the fact that it introduced logical clocks, a much less important idea” [Lampson 2018].

Xerox PARC had pioneered personal distributed computing in the early 1970s. Its Computer Science Lab (CSL) created what became the archetype of personal computing: the Alto workstation, the Ethernet, and essential network services for file storage and printing. In retrospect, it seems a natural fit for someone doing pioneering work in distributed systems, but at the time, Lamport didn't strike the leaders as sufficiently practically minded, so he didn't receive a job offer [[Lampson 2018](#)].

SRI was a different sort of research lab. Originally a part of Stanford University, SRI had become an independent entity doing contract research, much of it for the US government. When Lamport interviewed, they had a multiyear contract to build an airplane flight control system with stringent fault tolerance requirements, and they hired him to join that effort.

The SIFT project (software-implemented fault tolerance) originated indirectly in the so-called Arab Oil Crisis of 1973, in which an embargo on the export of oil from the Middle East caused prices to skyrocket in much of the world [[Wikipedia 2019a](#)]. Conservation efforts ensued, including research into techniques for making aircraft significantly lighter. To do so required active control of the flight surfaces, implying the need for automation to effect small adjustments many times per second. While computers performed a variety of functions in aircraft in the 1970s, those functions weren't critical, and the human crew could take over in the event of a failure. For aircraft requiring continuous adjustments to continue flying, a computer system became essential and obviously required high reliability, which could only be achieved through replication of critical components and software to manage those components coherently. NASA contracted with SRI to build that software. The contract specified a probability of system failure of less than 10^{-9} per hour, comparable with manned spacecraft systems [[Wensley et al. 1978](#)].

When Lamport arrived at SRI, much of the work on SIFT had been done, including the solution of a central problem: how to get a set of processors to agree on something in the presence of faults. Marshall Pease had proved that $3t + 1$ processors are necessary and sufficient to tolerate t faulty processors under customary communication assumptions.¹⁰ Lamport observed that the result could be significantly improved by the use of communication involving digital signatures, reaching back to some work he had done at COMPASS before digital signatures were generally known in the field. (They became known through Diffie and Hellman's landmark cryptography paper [[Diffie and Hellman 1976](#)], which appeared in

10. Pease was inspired by the work of his colleague down the hall, Robert Shostak, who had proved the result for $t = 1$, the case of practical interest for SIFT [[Shostak 2018](#)].

1976 but mentions Lamport's earlier digital signature work.¹¹) But Lamport says his main contribution to the paper describing the fault tolerance results [[Pease et al. 1980](#)] was getting Shostak and Pease to write the paper in the first place [[Lamport 2019](#), comment 41], as publication generally wasn't (and generally still isn't) a priority for researchers in industry.

Lamport continued to refine the fault tolerance algorithm (and its description), which led to a second paper with Shostak and Pease. The first paper eventually came to be recognized as a foundational result—it received the 2005 Edsger W. Dijkstra Prize in Distributed Computing—but the second paper gave the problem of agreement in the presence of arbitrary faults the name that has stuck. Lamport recalls being inspired by Jim Gray, who had earlier described a problem that he coined the “two generals paradox.” The problem involves two generals who want to coordinate an attack that cannot succeed unless both of their armies participate. The generals communicate only through messengers sent between them. The problem: Since any messenger may fail to arrive, how can the generals ever be certain that they have reached agreement to attack? The similarity with the fault tolerance problem of the SIFT project led Lamport to seek a similar catchy name [[Lamport and Levin 2016a](#), page 34, edited]:

I decided on a story of a bunch of generals who had to reach agreement on something and they could only send messengers and stuff like that. I originally called it the Albanian Generals, because at that time Albania was the most communist country in the world; it was a black hole, and I figured nobody in Albania is ever going object to that. Fortunately, Jack Goldberg, who was my boss at SRI said, “You really should think of something else because, you know, there are Albanians in the world.” So, I thought . . . and suddenly: Byzantine Generals! Of course, with the connotation of intrigue, that was the perfect name.

The Byzantine generals paper [[Lamport et al. 1982](#)] overlapped significantly with the earlier paper containing the central $3t + 1$ result. Lamport freely admits that “the main reason for writing this paper was to assign the new name to the problem” [[Lamport 2019](#), comment 46]. However, he reformulated the algorithm in a simpler way, since Pease's original algorithm, which Lamport characterized as “an amazing piece of work” [[Lamport and Levin 2016a](#), page 13], was difficult to

11. Diffie had been working on the digital signature problem in the mid-1970s. He met with Lamport in a coffeehouse in Berkeley and told him about the problem. Lamport recalls that he “thought a minute and literally on a napkin I wrote out a solution involving one-way functions” [[Lamport and Levin 2016a](#), page 14].

follow and the proof was even more so. As Lamport recognized, names matter, so while the earlier paper with the core result justly received the Dijkstra Prize, that result indelibly bears the name from the later one: Byzantine agreement. During his SRI years, Lamport continued to work on fault tolerance, incorporating the state machine formulation he introduced in the Time/Clocks paper. Chapter 3 discusses the original Byzantine generals result and subsequent ones.

Lamport's work at SRI extended well beyond fault tolerance. Much as at COMPASS, Lamport nominally worked on a variety of contracts, but actually carried out a largely self-directed research program, pursuing threads that reached back to earlier work and would continue well into the future. One such thread involved the arbiter problem, sometimes more colloquially named "the glitch," in which Lamport became interested some years earlier when it was noted that the Bakery algorithm required an arbiter. The problem arises from trying to decide between discrete outcomes—say, having a circuit output a zero or a one—in a bounded time interval. Engineers were familiar with the problem, though many did not accept that it was unsolvable. Lamport and Richard Palais, his former *de jure* thesis adviser, found a mathematical formulation using a carefully chosen topology that, coupled with theorems about continuity, produced a proof of impossibility. Their paper, submitted for publication in 1976, was rejected because, Lamport believes, the mathematics was foreign to the reviewers [Lamport and Levin 2016a, pages 24–25]. It is possible, however, that (in the spirit of "the impossible takes a little longer") they were reluctant to accept the conclusion. Lamport quotes a related story [Lamport 2019, comment 60]:

Charles Molnar, one of the pioneers in the study of the problem, reported the following in a lecture given on February 11, 1992, at HP Corporate Engineering in Palo Alto, California: "One reviewer made a marvelous comment in rejecting one of the early papers, saying that if this problem really existed it would be so important that everybody knowledgeable in the field would have to know about it, and 'I'm an expert and I don't know about it, so therefore it must not exist.'"

Lamport and Palais's paper was never published, but Lamport wrote a less formal treatment under the title "Buridan's Principle." Surprisingly, though it described the arbiter problem for a general scientific audience, neither *Science* nor *Nature* wanted to publish it. Eventually, in 2012, it found acceptance in *Foundations of Physics* [Lamport 2012], perhaps because that discipline was more comfortable with the mathematical ideas and with the sometimes unintuitive consequences of formalizing the behavior of physical systems.

During his SRI years, Lamport explored concurrency well beyond the arbiter problem. He contributed to an influential paper on concurrent garbage collection by Edsger Dijkstra in 1978. The paper was ahead of its time because, in the late 1970s, programming languages generally didn't depend on garbage collection and rarely ran on computer systems offering true concurrency. (Lamport considered his contribution minor, but Dijkstra thought it significant enough to make him a co-author.) In this area, as in others, Lamport foresaw practical problems and devised their solutions decades before most of the field recognized their need or importance. Other noteworthy papers by Lamport during this time include his work with Mani Chandy on distributed snapshots [[Chandy and Lamport 1985](#)] (see Chapter 2) and his paper on cache coherence [[Lamport 1979b](#)], which gives a precise definition of sequential consistency (see Chapter 1). As the size of programs increased, spurred on by the exponential growth of computing capacity characterized by Moore's law, many researchers sought techniques to verify program correctness. Since Lamport had worried about proving correctness at least since his COMPASS days, he naturally contributed to this area, focusing on suitable formalisms and techniques for verification of concurrent programs. He published several papers in this area while at SRI, including one that used arrows in a way reminiscent of the causality relation in the Time/Clocks paper, and others that represent initial forays into the use of temporal logic, which would become part of the methodology he ultimately adopted for specification and verification (see Chapter 5).

Aside from his Ph.D. thesis and a paper derived from it, Lamport never published a mathematical paper. But during his time at SRI, he published something that had a huge impact on the field of mathematics, as well as computer science: the \LaTeX system.

When Lamport began writing papers, computers provided only limited tools to assist authors of technical communications. Simple computer-based text processing systems had existed since the 1960s, but until the mid-1970s mechanical typesetting remained the technology for creating books and journals, even if the manuscript to be published had been prepared on a computer. Lamport himself did not begin using computer-based document production systems until around the time he arrived at SRI in 1977. What is now known as word processing was then in its infancy, appearing in experimental systems such as Butler Lampson and Charles Simonyi's Bravo [[Wikipedia 2018a](#)] and Brian Reid's Scribe [[Wikipedia 2018c](#)]. Coupled with computer-driven laser printing, these systems enabled authors to produce immediately publishable versions of technical documents, without going

through an intermediary that typeset their content. These new systems featured multiple fonts and text styles, hierarchical sectioning, footnotes, bibliographic citations, cross-referencing, and the like.¹² Lamport began using Scribe soon after it was available, around 1978 [Lamport 2018].

These early systems adopted an essentially linear document model that worked adequately for ordinary prose but could not accommodate the two-dimensional typesetting requirements of mathematics. Seeking to marry the benefits of an author-controlled technical document production system with the printing quality achievable in professional typesetting, Donald Knuth digressed from his ambitious book project, *The Art of Computer Programming*, to create a system capable of creating aesthetically pleasing documents with mathematical content.¹³ He named that system \TeX [Wikipedia 2019d].

\TeX made two-dimensional layout a first-class notion, which distinguished it not only from its contemporaries like Scribe and Bravo but also from most of their successors, which tend to treat formulas and other two-dimensional structures as a separate kind of object, like a picture or table, to be dropped into an otherwise linear text stream. Knuth also created the necessary fonts of mathematical symbols for formula construction.¹⁴ Finally, and crucially, Knuth wanted great flexibility in the typesetting, so he avoided building in assumptions about document structure. Instead, he gave \TeX a macro capability, enabling users to build structures for different purposes.

“If you build it, they will come.” The users came, and they built and exchanged macro packages, and \TeX quickly found adherents within the computer science and mathematics communities.¹⁵ Lamport began using it around 1979 for his papers. Knuth soon began working on a second version, which would come to be called $\text{\TeX}82$. Around the same time, Lamport began working on a book and decided that the available macro packages for something of that size were unsuitable, so he set

12. This was a classic example of technologists innovating to solve a problem they personally experienced: the cumbersome and error-prone publication cycle of manuscripts, galley proofs, and page proofs.

13. Knuth began this long-running book project in 1962. It was originally conceived as a single book of 12 chapters, then of 7 volumes, some of which now (in 2018) have several subvolumes.

14. Some mathematical symbols could be incorporated in documents produced in Bravo through its flexible font capability. Scribe could handle multiple fonts too, but in its early days it often ran in computing environments with printers of limited capability. Lamport recalls trying to create acceptable-looking mathematical symbols for dot-matrix printers and exploiting Scribe’s font capabilities to print them [Lamport 2018].

15. Other disciplines, including physics and statistics, came along later.

out to develop his own macros and thought, “I might as well make them usable for others” [Lamport 2018]. \LaTeX was born.¹⁶

\LaTeX combined the power of \TeX ’s 2D typesetting with Scribe’s notion of a document *style*. Scribe was one of the first document systems to provide a flexible and intuitive way of compartmentalizing detailed decisions about a document’s structure and appearance, a feature that Lamport wanted for his book project.¹⁷ Unsurprisingly, that proved a winning combination with many authors, and after a first version and manual became available around 1983, the system spread rapidly through the computer science community.¹⁸ The manual was later published by Addison-Wesley [Lamport 1994b]; the software itself has always been free.

Over the next few years, \LaTeX became the de facto standard way to create computer science papers and books, accelerating as personal computer workstations became more common. But with success comes support, and by the end of the 1980s a group of volunteers [\LaTeX Project 2019] had assumed responsibility for the evolution of \LaTeX , and Lamport ceased to be directly involved. (As of this writing (mid 2019), $\text{\LaTeX} 2\epsilon$, with a manual from 1995, is the current version. Version 3 is a long-running research project.) Three decades later, writers of computing papers with mathematical content, including Lamport, still use \LaTeX as their document production system of choice, though other systems, such as Microsoft Word, have caught up for other classes of papers.¹⁹

The book for which Lamport conceived \LaTeX , which he self-mockingly calls “The Great American Concurrency Book,” remains incomplete, though parts have

16. The Wikipedia article on \TeX [Wikipedia 2019d] explains its pronunciation (properly the final letter is pronounced as in Bach but often has the sound of a “k”). \LaTeX , Lamport’s system, is variously pronounced with a long or short “a”. Wikipedia doesn’t indicate which is preferred, but in ordinary conversation its creator generally says “lay-tek” [Lamport 2018].

17. In the first edition of the \LaTeX manual, Lamport freely acknowledged Scribe’s influence, quoting a sentiment attributed to Igor Stravinsky (among others): “Lesser artists borrow; great artists steal.”

18. The American Mathematical Society, which naturally embraced \TeX , initially standardized on a different set of macros called AMS- \TeX [American Mathematical Society 2019b], which existed in an early version when Lamport began work on \LaTeX . However, it did not include the larger-scale document structuring capabilities of \LaTeX . Subsequently, it harmonized the two in AMS- \LaTeX [American Mathematical Society 2019a].

19. In 2000 [Ziegler 2000], Lamport said: “I don’t think \TeX and \LaTeX would have become popular had they not been free. Indeed, I think most users would have been happier with Scribe. Had Scribe been free and had it continued to be supported, I suspect it would have won out over \TeX . On the other hand, I think it would have been supplanted more quickly by [Microsoft] Word than \TeX has been.”

been written more than once. There may be a parallel here with Knuth's *The Art of Computer Programming*.

6.5

The DEC/Compaq Years (1985–2001)

In 1985, Lamport went looking for another job. SRI had changed its management in a way that he felt introduced unnecessary structure, which often impedes research activity. As he looked for job opportunities, he sought a situation in which research would be stimulated by real-world problems, much as he had experienced while working on SIFT. He gravitated toward corporate research labs rather than academia for that reason, as professors in that era worked largely on self-defined problems supported by government funding agencies that took a more expansive view of research than they did a decade or two later [Lamport and Levin 2016a, pages 18–19]. Digital Equipment Corporation (DEC), a high-flying computer company in the 1970s and early 1980s, had recently opened a new lab in Palo Alto, the Systems Research Center (SRC), at which Lamport interviewed and was offered a position. Comparing SRC with SRI, he recalled [ibid., pages 17–18, edited]:

There was more community. At SRI, we said “Grant proposals are our most important product.”²⁰ The people who started SRC were the people who came from Xerox PARC, where they had just created personal computing, so there was very much a sense that the lab would continue to create the computing of the future. SRC had a qualitative feel that was different from SRI. That had a lot to do with Bob Taylor, the lab director and founder, who was just a wonderful manager.

Despite being young, SRC was already as large—about 15 researchers—as the group that Lamport had left at SRI, and within a year it had become considerably larger. Ultimately, it grew to about 60 researchers while retaining a completely flat management structure, one of Bob Taylor's many unconventional management practices.²¹ The absence of hierarchy made it easy for researchers to share research questions and to introduce each other to problems originating elsewhere in the company. In fact, as Lamport noted in retrospect, more of the problems on which

20. A parody of a contemporary General Electric slogan: “Progress is our most important product.”

21. Taylor came to DEC from Xerox PARC, where he had been the director of CSL. His disagreements there with his superiors over research management practices precipitated his departure and the founding of DEC/SRC.

he worked came from outside the lab, including especially from the rest of DEC [ibid., page 19].

By the early 1980s, the march of Moore's law had created single-chip microprocessors sufficiently powerful to be used for general-purpose computation. They were modest in performance but a great deal cheaper than the multichip CPUs that preceded them. The idea of using a multiprocessor to get affordable desktop computing power took hold in the research world. In 1984, as SRC was staffing up, DEC introduced its MicroVAX line, which used DEC's first single-chip CPU that implemented the (non-floating-point) VAX instruction set. Researchers at SRC launched a project to design a multiprocessor machine—the Firefly—around the anticipated second chip in the family.²² Since no standard operating system yet existed for multiprocessors, they launched a concurrent project to create one to work in harmony with Unix, which already was well established on the VAX line in addition to DEC's flagship operating system, VMS. Just down the street from SRC in Palo Alto, the Western Research Lab, a sibling DEC laboratory, was building another multiprocessor—the MultiTitan [DEC WRL 2018]—with a CPU chip based on WRL's earlier Titan [Nielsen 2018]. Both multiprocessor projects involved Lamport, though in different ways.

The hardware of a multiprocessor mediates access by the processors to the shared memory. This mediation depends on mutual exclusion, a topic that had engaged Lamport since his days at COMPASS. Shortly after Lamport came to DEC, the MultiTitan designers sought his help to create a mutual exclusion algorithm, using only reads and writes, that would be fast in the absence of contention. No one had considered this problem before multiprocessing became practical. Lamport created an algorithm and proved it was optimal [Lamport 1987].

Lamport's involvement in the Firefly project was less direct, though what he created would prove much more significant. SRC's network of Firefly workstations was to serve as a computing base for the researchers, many of whom had come from PARC. As such, it would consolidate what they had learned about building a practical, general-purpose, distributed computing environment. That environment would include scalable global naming, replicated fault-tolerant storage, and

22. This chip was colloquially called the MicroVAX II or Mayflower, though strictly speaking these names referred to the DEC product machine built around the 78032 CPU chip (and its companion 78132 floating-point unit).

network security; the project that put together those capabilities was the Echo Distributed File System [Birrell 1993]. Echo caught Lamport's attention, for its fault tolerance approach seemed to violate an impossibility result published a few years earlier [Fischer et al. 1985].²³ He recalled that "I sat down to try to prove that it couldn't be done. And, instead of coming up with the proof, I came up with an algorithm. . . . It's an algorithm that guarantees consistency,²⁴ and it gets termination if you're lucky" [Lamport and Levin 2016a, page 36]. That algorithm would come to be known as Paxos, and it was destined to become one of Lamport's most important inventions.

Lamport completed his writeup of the Paxos algorithm and its correctness proof in 1990 in a draft paper called "The Part-Time Parliament," though by then Echo was already in service using a different consensus algorithm to achieve consistency. (See Chapter 4 for a technical description of the Paxos algorithm.) The significance of Paxos was not widely recognized at the time. Lamport recalled [Lamport 2019, comment 122]:

Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island. Leo Guibas suggested the name *Paxos* for the island. I gave the Greek legislators the names of computer scientists working in the field, transliterated with Guibas's help into a bogus Greek dialect. (Peter Ladkin suggested the title.) Writing about a lost civilization allowed me to eliminate uninteresting details and indicate generalizations by saying that some details of the parliamentary protocol had been lost. To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist, replete with Stetson hat and hip flask. My attempt at inserting some humor into the subject was a dismal failure. People who attended my lecture remembered Indiana Jones, but not the algorithm. People reading the paper apparently got so distracted by the Greek parable that they didn't understand the algorithm.

23. Lamport said, "The FLP result, as it is generally known, says that, while you may have an algorithm in which multiple processes agree on a value, the algorithm cannot guarantee that such a value will eventually be chosen" [Lamport and Levin 2016a, page 36, edited]. He also characterized it as "one of the most, if not the most, important papers on distributed systems ever written" [ibid.].

24. The consistency guarantee means that the algorithm "can tolerate the failure of any number of its processes (possibly all of them) without losing consistency, and that will resume normal behavior when more than half the processes are again working properly" [Lamport 2019, comment 122].

When Lamport submitted the paper for publication, the reviewers wanted the Paxos story removed. Annoyed, Lamport put the paper aside.²⁵ Had it been taken more seriously, a reviewer might have noticed a strong similarity between the Paxos algorithm and one called viewstamped replication that appeared in Brian Oki's 1988 Ph.D. thesis [[Oki and Liskov 1988](#)]²⁶—a case of independent invention of which Lamport became aware only years later: “I looked in Brian’s thesis and there, indeed, was very clearly, the same algorithm. But I don’t feel guilty about it being called Paxos, rather than Timestamp [sic] Replication, because [Oki and Barbara Liskov, his adviser] never published a proof. And as far as I’m concerned, an algorithm without a proof is a conjecture” [[Lamport and Levin 2016a](#), page 37].

Meanwhile, the problem of maintaining consistency in replicated systems grew more important as networking and inter-networking became more widespread in the 1990s. At SRC, Ed Lee and Chandu Thekkath wanted to build a distributed system that provided replicated virtual disks, which they called Petal [[Lee and Thekkath 1996](#)]. They therefore needed a consistency protocol, and another SRC colleague, Mike Schroeder, directed them to the unpublished Paxos paper. They produced an implementation for Petal, apparently unhindered by the lost Greek civilization story. Subsequently, with SRC colleague Tim Mann, they built a scalable distributed file system called Frangipani on top of Petal, which used a different Paxos implementation for its distributed lock server.²⁷ With Paxos thus in use, in the mid-1990s Lamport was again motivated to try to publish his paper.²⁸ The revised version retained the storytelling of the original but added some annotations on related work from the intervening years, which were provided by Lamport’s colleague Keith Marzullo in a way that continued the parable. “The Part-Time

25. The anonymous reviewer wasn’t the only person who found the Paxos story problematic. Butler Lampson recalls: “When Leslie wrote the first paper about Paxos, it was pretty incomprehensible. . . . I had to read the paper about 6 times before I could figure it out, but then it was clear to me that it was really important.” Lampson tried unsuccessfully to convince Lamport to revise the paper [[Lampson 2018](#)].

26. The Petal and Frangipani implementations of the Paxos algorithm were partial; fuller implementations of the Paxos algorithm are discussed later in this chapter.

27. Butler Lampson recalls an additional motivation: “Leslie . . . refused to make any changes, so for eight years the paper remained unpublished. . . . I believe that what broke this logjam was that I wrote a description of the algorithm that Leslie hated” [[Lampson 2018](#)].

Parliament" finally appeared in 1998, nearly a decade after the invention of the Paxos algorithm.²⁸

During the years between the invention of Paxos and the eventual publication of the paper, Lamport continued to pursue three interrelated threads of research that had engaged him for more than a decade. The first, of course, was concurrent algorithms, including mutual exclusion, interprocess communication, and fault tolerance.²⁹ The other two topics—precise specification of system behavior and formal verification of program properties—had also prominently figured in his work, motivated initially by the experience that led to his discovery of the Bakery algorithm (recounted in Section 6.3). He published papers on the specification and verification of concurrent programs beginning in 1977 [[Lamport 1977a](#)], and by the time he came to SRC, he had published at least a dozen papers more. Much of this work sought to create practical methods that didn't collapse under the weight of real-world algorithmic problems.

By the time he invented Paxos, Lamport had created the temporal logic of actions (TLA), a formal logic for specifying the behavior of a state machine and proving invariance properties about it. He had been evolving toward this formalism since he introduced the state machine model in the Time/Clocks paper, and he had used invariance to characterize the meanings of programs even earlier at COMPASS in his work on parallelizing FORTRAN.

Lamport developed TLA at a time when many other researchers wanted to specify program behavior within a programming language. Lamport disagreed; he regarded mathematics as the correct vehicle. Accordingly, a specification in TLA could use mathematical objects not generally found in a programming language (such as sets), thereby freeing the specification from the clutter of implementation details

28. There is evidence that the original journal editor's objection to the style of the paper had some validity. Lamport reports [[Lamport 2019](#), comment 129] and Lampson confirms [[Lampson 2018](#)] that distributed systems specialists had difficulty understanding the algorithm. He then relented and published *Paxos Made Simple* [[Lamport 2001](#)] with a one-sentence abstract: "The Paxos algorithm, when presented in plain English, is very simple." At least one commercial implementation of Paxos took advantage of this paper [[Burrows 2019](#)], even though Lamport's web page warns against implementing from this informal description.

29. Around the time that Lamport joined SRC, he completed two two-part papers on mutual exclusion [[Lamport 1986a, 1986b](#)] and interprocess communication [[Lamport 1986c, 1986d](#)] that consolidated much of his work over the preceding decade or so. See Chapter 1. Considerably earlier, in a 1979 paper [[Lamport 1979b](#)], he had also considered the problem of multiprocessor cache coherence, a topic in mutual exclusion directly relevant to the Firefly.

extraneous to an algorithm’s essence. In short, TLA directly supported *abstraction*, an essential technique for managing the complexity of describing and verifying real-world systems. (See Chapter 5 for more background and details about TLA.)

TLA provided a formal way to state properties of programs, but proving those properties was another matter. Lamport knew the pitfalls of informal verification, and he recognized that for TLA to be useful for anything other than small examples, it would need to be supported by a mechanical proof system. Building a full-scale theorem prover was and is a substantial and daunting undertaking, so Lamport initiated a project to build a system to translate TLA formulas into the input of an existing theorem prover with which he and SRC colleagues had experience: LP, the Larch prover [Garland and Guttag 1988]. Armed with this tool—later developed further by Urban Engberg and dubbed TLP [Engberg 1996]—one could write a proof in TLA, translate it into the language of LP, and have LP check it.

However, TLA with TLP was an incomplete solution for Lamport’s purposes.³⁰ TLP demonstrated the feasibility of mechanically verifying a TLA specification, but it handled only a limited subset of what TLA could express. It was a step along the road, but many more would be required.

By 1993, Lamport had enough experience with TLA to begin using it in published work, introducing it first in a workshop paper [Lamport 1993] and then in a journal paper the next year [Lamport 1994c]. TLA figured prominently in most of his published papers over the next several years. Then, in 1996, TLA had its first opportunity to affect a DEC product.

By the time the initial version of TLP was showing promise, multiprocessing had moved from the research labs to the DEC product stream. During the 1980s, the “RISC versus CISC” controversy raged in the field, sparked by the work on Reduced Instruction Set Computers (RISC) at UC Berkeley (although similar architectures had existed earlier). Most commercial computers of the era were of the non-RISC type, at that time dubbed “CISC,” a back-formation for Complex (or Comprehensive) Instruction Set Computer. DEC’s flagship VAX architecture was CISC, but several influential engineers believed that the company’s survival depended on using a RISC design for the VAX’s successor. That architecture was eventually named

30. The initial work on TLP occurred around 1991, but the real-world test did not come until several years later when Georges Gonthier carried out a mechanical proof of a concurrent garbage collector for his Ph.D. thesis. Lamport wrote: “Gonthier estimated that using TLP instead of working directly in LP reduced the amount of time it took him to do the proof by about a factor of five” [Lamport 2019, comment 96].

Alpha [[Wikipedia 2019b](#)]. In the same spirit as the VAX and the IBM S/360/370 architectures, the Alpha design was intended to last 25 years, being implemented in different but functionally compatible ways as technology evolved. This meant, of course, that the Alpha architecture had to accommodate multiprocessor implementations and that the designers of each implementation would need to verify that it satisfied a common specification. That specification was the *Alpha Architecture Reference Manual*. It contained a major section on the Alpha memory model, which had been formally specified by three DEC researchers: Kourosh Gharachorloo at WRL and Jim Saxe and Yuan Yu at SRC.

In 1996, a DEC product team had designed a multiprocessor system called Wildfire that utilized a complicated Alpha processor with out-of-order execution, internally called EV6. The cache coherence protocol to support this processor was extremely complex. The protocol designers contacted Gharachorloo to inquire if he or perhaps other researchers would be interested in verifying that the protocol satisfied the Alpha memory model. Gharachorloo in turn contacted Yu and Saxe to see if they would participate. Yu forwarded the invitation to Lamport, who soon became involved [[Lamport 2018c](#)], along with Mark Tuttle of the DEC Cambridge (Massachusetts) Research Lab.

Yu and Lamport brought complementary skills to the EV6 protocol verification project. Yu's Ph.D. dissertation at UT Austin involved verification of low-level code and had been supervised by J Moore, one of the legends of mechanical theorem-proving. Lamport, in addition to his nearly two decades of experience with precise specification and correctness proofs, had by this time created TLA⁺, an extension of TLA with capabilities to support large formulas,³¹ as well as a methodology for organizing large proofs hierarchically [[Lamport 1995](#)], both of which would be essential for the EV6 verification effort.

Essential, yes, but barely sufficient. TLA⁺ lacked tools to support writing or checking specifications, so the work was entirely manual. Lamport, Tuttle, and Yu began by studying the EV6 documents, consulting the hardware engineers in the product team when necessary to understand the protocol's detailed workings. They wrote in TLA⁺ both an invariant describing the Alpha memory system and an abstraction of the protocol that EV6 used to implement it, then proved that the protocol indeed preserved the invariant.

³¹ The definition of TLA⁺ didn't appear in broadly available form until 2002 [[Lamport 2002](#)], though an earlier version of part of that book appeared in 1999 in course notes for a summer school [[Lamport 1999](#)]. See Section 5.3 for an overview of TLA⁺.

Of course, it wasn't nearly that simple; the process was highly iterative and took most of a year. Yu had no experience in formulating invariants of this kind. He recalled that they spent most of the time in Lamport's office, proposing abstractions and invariants and trying to prove that one implied the other. Usually, an invariant was either wrong or too weak, so the proof couldn't be completed. Even when it could be, they would often discover later it was flawed, since they had no tools to check it rigorously [Yu 2018]. To carry out a complete proof by hand would have been much too time consuming so, as they wrote [Lamport et al. 2002]:

We selected two conjuncts, each about 150 lines long, as the part of the invariant most likely to reveal an error. We completed the proof for one of the conjuncts; it was about 2000 lines long and 13 levels deep. The proof of the second conjunct would have been about twice as long, but we stopped about halfway through because we decided that the likelihood of its discovering an error was too small to justify further effort. We spent about seven months on these two proofs. We also wrote an informal higher-level proof of one crucial aspect of the protocol. It was about 550 lines long and had a maximum depth of 10 levels.

While it was now evident that TLA^+ was equal to the task of specifying a real-world memory system, neither Yu nor Lamport cared to attempt another such project without tools. Yu began working on a model checker for (a subset of) TLA^+ , though Lamport was skeptical that it could be done and even tried to talk Yu out of it. Yu persevered and within about three months had a working proof-of-concept model checker. Lamport was immediately convinced and became a strong supporter of TLC, as the model checker for TLA^+ came to be called. (See Section 5.5.1.) Other than a simple syntax checker, TLC was the first real tool supporting the production of TLA^+ specifications.³² It was none too soon, for a few months later, Lamport and Yu would undertake the verification of EV7, the next chip in the Alpha line.

The EV7 verification began in early 1998. "This time, the specification was written by [a DEC product] engineer who received a few hours' instruction on TLA^+ . (At the time, there was no language manual.) His specification was about 1800 lines long" [ibid.]. Though no fanfare accompanied it, this marked the first use of TLA^+

32. A model checker is not a theorem prover. A theorem prover, when it succeeds, demonstrates that a theorem is correct, but often gives little guidance when the "theorem" is wrong. By contrast, a model checker examines a purported theorem in a specific, concrete instance (e.g., for a memory system, the case of exactly two processors) and looks exhaustively for a counterexample. Model checking is far more useful during the development and debugging of a specification, when it is nearly always wrong, since the model checker pinpoints each error.

by someone outside a research organization. The product group put TLC to work and it paid off [Lamport et al. 2002]:

As soon as we started using TLC, we found many errors in the TLA⁺ specification. Not counting simple mistakes that were easily corrected, we found about 70 errors. About 90% of them were discovered by TLC; the rest were found by a human reading the specification. Most of the errors were introduced when translating from the informal specification; they demonstrate the ambiguity inherent in such specifications. Five design/implementation errors were discovered—one directly by TLC, the other four by using TLC error traces to generate simulator input.

The Alpha product engineers were pleased with the outcome. They planned to use TLA⁺ and TLC for EV8, the next processor in the Alpha line, and to replace the English version of the cache coherence protocol specification with the TLA⁺ one. But though TLA⁺ was in the ascendant, DEC and the Alpha were not. Compaq acquired DEC in 1998 and had no use for Alpha. The architecture that was intended to last 25 years was sold to Intel in 2001, and DEC engineers went with it. Among them were some who continued to use the TLA⁺ approach to specification, marking the first use of TLA⁺ without Lamport as an in-house consultant.

In the several years preceding the EV6 verification project, Lamport had developed a style of proof well suited to the theorems that arise in formal verification. As the preceding quotations indicate, these formulas are large—the specification for the EV6 and EV7 cache coherence protocols each amounted to about 1800 lines. The conventional proof style favored by mathematicians, being informal and essentially linear, cannot cope with such complexity. As Lamport observed, “The structure of mathematical proofs has not changed in 300 years. The proofs in Newton’s *Principia* differ in style from those of a modern textbook only by being written in Latin. Proofs are still written like essays, in a stilted form of ordinary prose” [Lamport 1995].

Conquering that complexity required a methodological tool that Lamport knew well from his background as a programmer: hierarchical decomposition. Lamport would repeatedly divide the steps of a proof into smaller and smaller statements to be proved, essentially in the style of a formal deduction of a statement in mathematical logic, familiar to anyone who has taken a first-year symbolic logic course. How far should that iterative decomposition go? Lamport wrote: “My own rule of thumb is to expand the proof until the lowest level statements are obvious, and then continue for one more level. This requires discipline” [ibid.]. The extent of

that discipline required for the EV6 verification is evident from looking back at the numbers quoted above.

Discipline in carrying out a hand proof is a good thing, but relying on proof automation is better still. “Structured hand proofs are much more reliable than conventional mathematical proofs, but not as reliable as mechanically checked ones” [Ladkin et al. 1999]. For TLA^+ to be more broadly adopted, Lamport would need to provide it with tools that reduce the amount of discipline required.³³ That would be a priority in the next phase of his career.

6.6

The Microsoft Years (2001–)

In early 1996, Bob Taylor retired as the director of SRC, unhappy with management changes that occurred as DEC attempted to adapt to the Internet age. Roy Levin took over as director of SRC. Within two years, Compaq Computer Company, which had established itself as a leader in the so-called IBM PC clone business, acquired DEC, but was unable to capitalize on DEC’s technical and market strengths. As support for research eroded, Levin and Assistant Director Mike Schroeder approached Rick Rashid, founder and head of Microsoft Research, with the offer to create a lab in Silicon Valley. Rashid was supportive, and in August the lab opened. By the end of 2001, a few researchers from DEC/Compaq had joined, and Lamport was among them.

Before moving to Microsoft, Lamport had been devoting much of his attention to TLA^+ and its associated tools. The DEC engineers who moved to Intel with the sale of the Alpha in 2001 wanted to continue using TLA^+ , and therefore wanted a license agreement to use Compaq’s intellectual property. The sale of Alpha had been a symptom of Compaq’s decline, and in September 2001, HP announced that it would acquire Compaq.³⁴ Shortly before that announcement, Compaq released TLA^+ code under a BSD-style license, which enabled Lamport to continue to evolve it at Microsoft and to share the results broadly.

Lamport wanted to continue the TLA^+ collaboration with the now Intel engineers, but he also saw opportunities for TLA^+ within his new employer. DEC had been chiefly a hardware company, even though it did significant software development. Microsoft had been a software company since day one and had a vast array

33. Indeed, the need for hierarchical proof structure accompanied by mechanical verification had already been noted in the first use of TLA with the TLP prover, described earlier.

34. 2001 was also the year that Dell moved ahead of Compaq as the biggest supplier of PCs, a fact that certainly influenced HP’s decision to acquire Compaq [Wikipedia 2018b]. The HP acquisition of Compaq was completed in mid-2002.

of software systems, both products and internal services. Many of these systems were just beginning to come to grips with distributed computing environments and the problems of concurrency in which Lamport was an expert. An outside observer might have thought it an ideal marriage of skills and needs.

Nevertheless, the relationship progressed slowly. A new research laboratory with ten or twenty researchers needs time to find its way in a corporation with tens of thousands of software developers, despite the well-established mechanisms within Microsoft Research for engaging with product organizations. The Silicon Valley lab sat on a satellite campus, and as most of Microsoft's development work occurred in the corporate center in Redmond, Washington, product organizations and researchers had to learn how to build relationships at a distance. The ex-SRC researchers knew all about this, since DEC's central development sites had been across the country in Massachusetts, but it was a new way of working for product developers accustomed to having researchers a few blocks away. Even though the new lab was in the same time zone—an advantage that SRC researchers hadn't had—it took some time before product groups became comfortable with remote collaboration. Unsurprisingly, researchers at the new lab established initial ties to product groups on the satellite campus in Mountain View where the new lab resided. However, none of these connections provided a natural outlet for Lamport's expertise.

Once the TLC model checker had demonstrated its value in hardware verification, Lamport knew that TLA^+ now could reach a wider user audience. Writing formal specifications without tools required dedication that only Lamport and his closest disciples could muster; with tools, the methodology could be adopted by practitioners and taught to a new generation of engineers. At DEC, Lamport had begun to promulgate the TLA^+ approach—he had begun writing a book on concurrency before the Compaq acquisition and had used portions of it in a course in 1998 [Lamport 1999]. At Microsoft, he completed the book *Specifying Systems* and published it in 2002 [Lamport 2002]. By this time, the TLA^+ toolset comprised the TLC model checker, the syntactic analyzer that parses specifications and checks for errors, and the TLATEX typesetter that renders the plaintext specifications accepted by other tools more readable using established publication conventions for mathematical works.³⁵ Lamport explicitly described *Specifying Systems* as the reference manual for TLA^+ and its tools, and much of it is devoted to extensive, varied examples that illustrate Lamport's approach to formal specification. To reduce the

35. Leveraging Lamport's own LATEX, of course.

adoption barrier further (and with the notable support of his publisher, Addison-Wesley), Lamport made the book freely available online for noncommercial use.

With this material now readily available, Lamport pursued a two-pronged strategy for TLA^+ : engage with product organizations to help them understand how TLA^+ can benefit them and enhance the toolset to make TLA^+ more robust and more attractive to practitioners. He was under no illusions about how difficult the first of these would be. In 2002, he published an experience paper [[Lamport et al. 2002](#)] with coauthors who had joined him on specification/verification projects at DEC. In it they wrote [page 4]:

An important lesson we have learned is that moving formal methods from the research community to the engineering community requires patience and perseverance. Engineers are under severe constraints when designing a system. Speed is of the essence, and they never have as many people as they can use. Engineers must be convinced that formal methods will help before they will risk using them.

They also noted that, although there is “no fundamental difference between hardware and software” when dealing at the specification level, hardware engineers are both more accustomed and more inclined to use formal methods for specification, in part because of the higher, more immediate cost of errors in hardware design. They further noted:

TLA^+ , which has proven useful for hardware, should be just as useful for software. However, there does seem to be a cultural difference between hardware and software engineers. Software engineers do not have the same tradition of relying on specifications that hardware engineers do.

Now that Lamport worked in a software company, he knew he faced a challenging path to get TLA^+ adopted for specifying software systems.

By late 2002, when *Specifying Systems* appeared, the so-called dot-com boom had ended rather abruptly, and the industry, perhaps somewhat chastened, began regrouping. The term “Web 2.0” became popular, loosely referring to web-based behavior beyond the first generation of static web pages. Rather than being a specific technical idea, Web 2.0 was a philosophy of interaction between users and services that emphasized the more dynamic behavior now familiar in such ubiquitous capabilities as online shopping and forums. Such a world required communication capabilities and standards beyond the basic HTTP protocol of the early web.

Microsoft, originally successful as a packaged software company, was rapidly retooling itself to address this new distributed world. It had several organizations focused on “web services.” Lamport and a visiting academic colleague, Friedrich “Fritz” Vogt, met two key architects in Redmond, James Johnson and David Langworthy, who liked the idea of creating formal specifications for web services protocols for which they had informal descriptions. Lamport summarized the resulting collaboration to write a specification for a web services atomic transaction protocol [Lamport 2019, comment 150]:

Fritz and I spent part of our time for a couple of months writing it, with a lot of help from Jim and Dave in understanding the protocol. . . . This was a routine exercise for me, as it would have been for anyone with a moderate amount of experience specifying concurrent systems. Using TLA⁺ for the first time was a learning experience for Fritz. It was a brand new world for Jim and Dave, who had never been exposed to formal methods before. They were happy with the results. Dave began writing specifications by himself, and has become something of a TLA⁺ guru for the Microsoft networking group.

The experience paper reporting on this project appeared in 2004, by which time Lamport’s other efforts toward TLA⁺ adoption within Microsoft had begun to pay off. Notably, he was no longer the only person using TLA⁺ within the research organization: Tom Rodeheffer, a local colleague, was engaged with another product group designing a protocol, and two researchers in Microsoft’s Redmond lab were using TLA⁺ to specify the protocol for a distributed storage project. Dave Langworthy’s initial enthusiasm had matured into advocacy, noting that “[My team and Lamport] spent a day working on replication and discussed several other topics. . . . Above the specific interactions, Leslie has improved the way we think about building distributed systems” [Langworthy 2005].

Any builder of software knows that, regardless of how well developed the concept behind a system is, its users always ask for more. So, as Lamport pursued the second prong of his TLA⁺ strategy—growing the toolset—he enhanced the system in response to his users’ experience and requests. Yuan Yu, who had built the original TLC, reimplemented it more robustly and with somewhat expanded capabilities. But though the model checker made specifications easier to debug, it didn’t help with writing them in the first place.

One frequent obstacle for engineers encountering TLA⁺ was its grounding in mathematical logic, a subject superficially related to programming but essentially different. This difference created a wide gulf between the way engineers think

about algorithms and the way they are specified in TLA^+ . To bridge that gulf, Keith Marzullo suggested creating a form of pseudocode, which engineers often use to sketch algorithms, that could be mechanically translated into a TLA^+ specification. Lamport and Marzullo built the translator and called their pseudocode language PLUSCAL (initially, +Cal), a nod to Pascal, which had often been used as a basis for informal pseudocode³⁶ [Lamport 2009, 2006c]. (See Section 5.4 for an overview of PLUSCAL.)

Even with the addition of PLUSCAL, the TLA^+ Toolbox, as it was now called, still lacked an important capability: mechanical proof. TLC, the model checker, had made it practical to find errors in specifications, but as Edsger Dijkstra famously observed, “Program testing can be used very effectively to show the presence of bugs but never to show their absence” [Dijkstra 1971]. Since the experiments with TLP in the early 1990s (see Section 6.5), Lamport had wanted a theorem prover for TLA^+ . In 2006, he found a way to get one. Microsoft Research and INRIA had recently launched a joint research lab in Paris, and several of Lamport’s former collaborators began working there. He organized a project to create a modern theorem prover for TLA^+ . Despite advances in the theorem-proving technology and vastly more powerful hardware on which to run it, the project faced considerable challenges in producing a usable tool. In 2010, the first release of TLAPS, the TLA^+ Proof System, joined the TLA^+ Toolbox [Microsoft Research 2019]. (See Section 5.5.2 for an overview of TLAPS.)

The arrival of the theorem prover amplified the efforts of Lamport’s colleagues who were trying to use TLA^+ for their own research. Chief among them was Tom Rodeheffer, who had been intrepid enough to use TLA^+ even before the model checker was available. Now Rodeheffer was able to apply TLA^+ to a partial replication system, a distributed atomic memory system, and a data-center network built out of configurable switches. Rodeheffer sat two offices away from Lamport and was the source of many suggestions for enhancement of the tool suite.

The last component of the Toolbox differed in character from its predecessors. The word “toolbox” suggests something holds all the tools, which perform their individual functions. Indeed, from the outset, the individual TLA^+ tools had been

36. Lamport has often written about the way in which programming languages limit software developers’ thinking, and he repeatedly resisted (indeed, railed against) adding language-like features, such as types, to TLA^+ . PLUSCAL doesn’t do that; it provides a thin layer of surface syntax over TLA^+ to help those who, Lamport might say, are afflicted with “Whorfian syndrome,” which he defines as confusing language (programming) with reality (mathematics) [Lamport and Levin 2016a, page 22].

built in the conventional command-line style. While natural enough in the previous century, in the present one the lack of tool integration was a bit quaint. Developers had increasingly become accustomed to using a programming environment in which the tools were tightly integrated, so that program editors, compilers, debuggers, performance analyzers, and the like seamlessly shared an understanding of the program under development and could display its various aspects simultaneously in a collection of interrelated display windows. TLA^+ needed a similar interactive development environment (IDE).

To build the IDE for TLA^+ , Lamport turned to Simon Zambrovski, who spent a postdoctoral year working with Lamport at Microsoft. He built it on Eclipse, a widely available open-source platform used chiefly for Java, in which the original TLA^+ tools had been written. The IDE, first released in 2010, received subsequent improvements from Markus Kuppe (who as an intern with Lamport had built a distributed version of TLC that enabled it to check specifications on larger cases) and Daniel Ricketts. By 2013, TLA^+ had an integrated, comprehensive Toolbox available to the computing community [Lamport 2019]. (See Section 5.5.3 for an overview of the Toolbox.)

Finally, more than two decades after the creation of TLA , Lamport could see the fruits of his efforts. TLA^+ and its toolset had taken root in Microsoft product organizations and in other companies. Brannon Batson, a former DEC engineer on the Alpha team that went to Intel, characterized TLA^+ around 2006 as follows [Merz 2008]:

The next big frontier in computer engineering is algorithmic complexity. In order to tackle this increasingly complex world, we need tools and languages which augment human thought, not supplant it. TLA^+ is a language which connects engineers to the underlying mathematics of their design—providing insight which they otherwise wouldn't have.

Later, Chris Newcombe, a former principal engineer at Amazon, wrote on a TLA^+ discussion group [Newcombe 2012]:

TLA^+ is the most valuable thing that I've learned in my professional career. It has changed how I work, by giving me an immensely powerful tool to find subtle flaws in system designs. It has changed how I think, by giving me a framework for constructing new kinds of mental-models, by revealing the precise relationship between correctness properties and system designs, and by allowing me to move from “plausible prose” to precise statements much earlier in the software development process.

By 2012, TLA^+ was in regular use at Amazon, with the distributed model checker running on hundreds of computers [ibid.]. Lamport believes that Amazon's experience, reported in [Newcombe et al. \[2015\]](#), was influential in stimulating use of TLA^+ within Microsoft's Azure team.

Much of Lamport's effort on TLA^+ in his Microsoft years could be called the unglamorous part of research: taking an idea that has had a successful proof of concept and pushing it forward into a practical system that others can and want to use. The effort required to do this in a research environment should not be underestimated, as the innovator cannot usually bring an idea to fruition alone and colleagues with the right skills and temperament for this kind of work are rare. Lamport was very fortunate to find such colleagues in the research labs at DEC, Microsoft, and INRIA.³⁷ Nevertheless, getting TLA^+ adopted for precise, verifiable specification of algorithms remains one of Lamport's long-term projects.

While maintaining a steady pressure to advance TLA^+ and its toolset, Lamport also pursued other research interests. During the same years in which the TLA^+ Toolbox was expanding, Lamport continued to work on the problem of agreement protocols that had inspired his Paxos work in the 1990s. Even before the Paxos paper appeared in 1998, a product effort at DEC to build a replicated storage system had led Lamport and colleague Eli Gafni to create Disk Paxos [[Gafni and Lamport 2003](#)]. Within a few years, the explosion of Internet-based services led various groups at Yahoo, Google, Microsoft, Facebook, and elsewhere to build and deploy systems that used Paxos (see Section 4.6.2). However, these systems did not use the full generality of the algorithm.

Lamport's insights into the fundamental behavior of distributed systems had led him, in Paxos, to specify a very general algorithm that intertwined the solutions to two problems: replication of state to achieve robustness and reconfiguration to maintain system availability. In practice, this intertwining posed many implementation challenges, which is why Paxos implementations from Petal onward used Paxos's state machine replication but invented alternative, simpler mechanisms for availability. In the mid-2000s, Lamport worked on variants of Paxos that improved its performance by reducing the number of replicas (Cheap Paxos [[Lamport and Massa 2004](#)], inspired by Microsoft product engineer and coauthor Mike Massa) or messages required for the protocol (Fast Paxos [[Lamport 2006b](#)]), but these

37. In an interview in 2002, Lamport was asked: "To what extent do you consider research fun versus hard work?" He replied: "Hard work is hauling bales of hay or cleaning sewers. Scientists and engineers should be grateful that society is willing to pay us to have fun" [[Milojicic 2002](#)].

enhancements did not directly address the complexity of implementing reconfiguration. Eventually, however, colleague Dahlia Malkhi persuaded Lamport that the separation of replication and reconfiguration, which occurred in all known Paxos implementations, needed a principled foundation. At the end of the decade, Lamport, Malkhi, and their colleague Lidong Zhou published Vertical Paxos [Lamport et al. 2009a] and two related papers [Lamport et al. 2009b, Lamport et al. 2010] that established that foundation (see Section 4.7).³⁸

During these years of Paxos evolution, Lamport also continued to explore the relationships between Paxos and other agreement protocols, including a paper with Jim Gray showing that classic two-phase commit is a special case of Paxos [Gray and Lamport 2006] and a paper showing the Castro-Liskov algorithm for handling Byzantine faults is a refinement of Paxos [Lamport 2011]. The latter paper notably exploited the TLA⁺ Proof System, TLAPS, to formally verify the relationship.

By 2012—more than two decades after Lamport invented the Paxos algorithm—its importance for creating practical, large-scale, fault-tolerant distributed systems had been widely recognized, and “The Part-time Parliament”—a paper that almost was never published—won the SIGOPS Hall of Fame Award.³⁹

As of this writing (mid 2019), Lamport continues his research at Microsoft. Though Microsoft Research Silicon Valley was closed in 2014, he collaborates with colleagues at INRIA and elsewhere. For someone who has spent his entire career in industry rather than academia, Lamport invests very considerable effort in education through both frequent lectures and papers. Indeed, roughly a tenth of his papers could be viewed as efforts to teach others what he has learned through his research rather than as reports of that research. Unsurprisingly, the fraction of such papers has increased in recent years; after nearly half a century of carrying out and reflecting on research, he increasingly tries to distill in writing the insights he has gained. He believes in the power of writing; in *Specifying Systems*, he opens with a favorite quote, from cartoonist Dick Guindon: “Writing is nature’s way of letting you know how sloppy your thinking is.” Some examples of Lamport’s own pithy observations:

38. Vertical Paxos provided a jumping-off spot for researchers other than Lamport to explore further evolution of the algorithm, notably Howard, Malkhi, and Spiegelman’s Flexible Paxos [Howard et al. 2016].

39. Somewhat analogously, a paper by Butler Lampson and Howard Sturgis, “Crash Recovery in a Distributed Data Storage System,” also won a SIGOPS Hall of Fame Award, but was never published [Lampson and Sturgis 1979]. For years, it was informally referred to as an “underground classic.”

- On using mathematics rather than natural language for specifications: “Mathematics is nature’s way of letting you know how sloppy your writing is” [Lamport 2002, page 2].
- On the indispensability of proof: “Never believe anything that is obvious until you have a proof of it” [Lamport and Levin 2016a, page 33].
- On using structured rather than prose proofs: “A proof should not be great literature; it should be beautiful mathematics. Its beauty lies in its logical structure, not its prose” [Lamport 2012, page 19].
- On distributed systems: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable” [Lamport 1987]. This is probably his most frequently quoted aphorism, written in an email in 1987.

6.7 Honors

Lamport’s work has been extensively recognized:

- National Academy of Engineering (1991)
- PODC Influential Paper Award (2000) (for [Lamport 1978b])
- Honorary doctorate, University of Rennes (2003)
- Honorary doctorate, Christian Albrechts University, Kiel (2003)
- Honorary doctorate, École Polytechnique Fédérale de Lausanne (2004)
- IEEE Piore Award (2004)
- Edsger W. Dijkstra Prize in Distributed Computing (2005) (for [Pease et al. 1980])
- Honorary doctorate, Università della Svizzera Italiana, Lugano (2006)
- ACM SIGOPS Hall of Fame Award (2007) (for [Lamport 1978b])
- Honorary doctorate, Université Henri Poincaré, Nancy (2007)
- LICS 1988 Test of Time Award (2008) (for [Abadi and Lamport 1991])
- IEEE John von Neumann Medal (2008)
- National Academy of Sciences (2011)
- ACM SIGOPS Hall of Fame Award (2012) (for [Lamport 1998a])
- Jean-Claude Laprie Award in Dependable Computing (2013) (for [Lamport et al. 1982])

- ACM SIGOPS Hall of Fame Award (2013) (for [[Chandy and Lamport 1985](#)])
- 2013 ACM A. M. Turing Award (2014)
- American Academy of Arts and Sciences (2014)
- Jean-Claude Laprie Award in Dependable Computing (2014) (for [[Wensley et al. 1978](#)])
- Edsger W. Dijkstra Prize in Distributed Computing (2014) (for [[Chandy and Lamport 1985](#)])
- Honorary doctorate, Brandeis University (2017)
- Fellow, Computer History Museum (2019)

It is noteworthy that the first of these honors—membership in the National Academy of Engineering—is one more customarily awarded considerably later in an individual’s career, after other honors have brought them to prominence. One would also be hard pressed to identify another computer scientist with as many honorary doctorates, even among those in the elite category of Turing Award recipients.

Several awards Lamport received recognize work that has stood the “test of time.” While such awards inherently occur years after the work they recognize, in Lamport’s case recognition occurred after uncharacteristically long intervals. The Time/Clocks paper [[Lamport 1978b](#)] appeared in 1978; it was recognized in 2000 and 2007. The Byzantine generals work appeared in papers in 1980 and 1982; it was recognized in 2005 and 2013. The Paxos paper, published in 1998 but embodying work done nearly a decade before, was recognized in 2012. And the Distributed Snapshots paper [[Chandy and Lamport 1985](#)] that Lamport wrote with Mani Chandy, another luminary in the field of distributed systems, appeared in 1985 and was recognized in 2013 and 2014.⁴⁰ These intervals, averaging roughly 20 years, suggest how far-sighted some of Lamport’s most important works have been and how long it took the field to fully appreciate them, since for most awards the “test of time” is a decade. (To be fair, some of the awards only came into existence in the 21st century, though Lamport’s seminal works weren’t always recognized in their inaugural year.)

Lamport received the ACM A. M. Turing Award, generally considered the equivalent of the Nobel Prize in computing, for 2013. It recognizes a body of work “for major contributions of lasting importance to computing.” Lamport’s award citation reads: “For fundamental contributions to the theory and practice of distributed

40. Surprisingly, this was the only paper these two distributed computing experts ever wrote together.

and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.” While the specific concepts listed occupied much of Lamport’s attention in the 1970s and 1980s, the first portion of the citation embraces his work on Paxos and fault tolerance as well.

6.8

Collegial Influences

As of this writing, Lamport’s “My Writings” web page [[Lamport 2019](#)] lists 184 works.⁴¹ Fewer than a third of them (59) have coauthors, a fraction that would be unusual for any computer scientist. Some people attribute this to prickliness, claiming Lamport is hard to work with. Obviously, those who have succeeded view the matter differently, painting instead a picture of a serious and uncompromising yet supportive collaborator. Their experiences in working with Lamport, related below with minor editing for readability, also provide additional insight into Lamport’s modus operandi and its impact on his colleagues.

Lidong Zhou [[Zhou 2018](#)]

Working with Leslie serves as a constant reminder of what scientific research is all about. Leslie has always been a role model who demonstrates how a scientist is driven by curiosity, stays insulated and focused in the pursuit of truth and perfection, and sticks to the core principles of research, which are often forgotten as too many of us constantly rush to get papers published in top conferences or journals as a means to gain personal fame.

Mani Chandy [[Chandy 2018](#)]

Leslie is a master at identifying and specifying core problems that abstract the essence of a myriad of practical situations that appear to be different.

You can see the nuggets, and their impact across a spectrum of applications, throughout Leslie’s career. Leslie specified the key idea formally, using mathematics; however, the path to get to the key idea used anthropomorphisms and analogies. A lesson from Leslie’s career for future generations of computer scientists is the importance of identifying core problems and specifying them clearly. Identifying the nugget is as important as finding the solution.

41. Nearly all these publications pertain to computing, but two show another side of Lamport as a scientist: “On Hair Color in France” [[Gilkerson and Lamport 2004](#)] and “Measuring Celebrity” [[Lamport 2006](#)]. Both appear in the *Annals of Improbable Research*.

Yuan Yu [Yu 2018]

All of Leslie's work is derived from practical systems. His ability to gain insight into the system problems and be able to abstract them out—I haven't seen anyone else who can do this.

My ten months of work with him on TLC was like an intensive training camp. Unfortunately, not everyone can have this kind of experience! He was not treating the EV6 specification as tutoring—we were partners, working on a project.

I don't think I mastered the ability to abstract problems. Leslie is still the master, but I got part of it, and that's very valuable to me. Later on, in all the systems I was involved in building, I was thinking about them in terms of state and the inductive invariants. I'm not sure that this is the only way, but this is clearly one very productive way of thinking about systems.

Some people say Leslie is hard to work with, but obviously I didn't have that experience. He tolerated my inexperience; I'm very grateful. He does have a very low tolerance of nonsense. . . . If you talk about something that he thinks is clearly nonsense and you have an argument with him, that's when he's going to show his anger.

Fred Schneider [Schneider 2018]

Lamport and I first met in person at the December 1979 SOSP conference at Asilomar. We got along, which is not something you can ever explain—simply a matter of personal chemistry. I would observe that we both grew up in New York City in working-class Jewish families (though neither of us was religious). Not only did we both have interest in fault tolerance, but we both were interested in reasoning about concurrent programs, which is quite a separate subject with a disjoint community of researchers. Most importantly, though, I think we both were less interested in building artifacts than in identifying principles.

Writing our first joint paper ("The Hoare Logic of CSP, and All That") was quite the education for me. He sat at a terminal; I sat next to him. He'd type a sentence, I'd read it, and we fought about wording. And notation. And formatting. I learned a good deal about writing technical papers from this experience, though at times it did get tiresome. Lamport had a strong idea about where the paper was headed, and he had developed the precursor to the programming logic we were developing as we wrote. I understood neither very deeply. But this meant that Lamport, sitting there, got to see "in real time" how a naive reader might misunderstand the points he wanted to make and the exact wording, especially since I was viewing things through the lens of the Owicky-Gries method, which was the defining characteristic of our target audience. . . . I don't recall how many visits I made to the West Coast,

but the paper was written in these five-day sessions of sitting side by side. (Two of our subsequent . . . papers were written in this way too. Needless to say, finishing a paper took quite some time, but the prose was well exercised.

Butler Lampson [Lampson 2018]

When Howard Sturgis and I wrote our paper on crash recovery in a distributed storage system [Lampson and Sturgis 1979], we never published it because we couldn't figure out how to prove the correctness of the algorithm, so it became known only in samizdat.

Like almost everyone else who does it seriously, I learned how to do proofs of concurrent systems from Leslie.

Stephan Merz [Merz 2018]

Leslie tends to be initially skeptical of ideas he hasn't come up with, but he is quick to realize a good idea, adopt it, and develop it further. Several years ago, I had a student who worked on a variant of the PLUSCAL language and suggested some extensions that would allow more elaborate algorithms to be modeled in it. Among the extensions was what we considered to be a minor suggestion for adding fairness annotations to processes and to labels within processes. When Leslie heard about this work, he was furious because we had dubbed the language "PLUSCAL 2" for lack of a better name. When we explained that this was just an internal moniker, he agreed to discuss the proposed extensions during a visit to Paris. He considered that most of our proposed additions were irrelevant or contrary to the spirit of PLUSCAL. However, he told us that he would steal the fairness annotations—except that prefixing a label by "fair" or "strong fair" (as we had done) was too heavy a notation, and he would rather affix a single character to the label. This is now part of PLUSCAL. Of course, our extended language is long forgotten. Incidentally, this anecdote shows Leslie's strong ideas about notation that permeate his work.

Tom Rodeheffer [Rodeheffer 2018]

I had a couple of trips to Paris (to theorem-proving conferences). I was there once visiting the INRIA lab and Leslie was there at the same time. He had me over for lunch at his apartment there and that was pretty cool because we just walked to his apartment and had a simple lunch of bread and cheese. The cheese was the best I had ever had so I asked about it. Leslie described the cheese and then told me all about the French cheese stores that I had never known to exist! He's a very personable guy, and here I am—this random person that's just using his tools, and he was very friendly.

Dahlia Malkhi [Malkhi 2018]

I am awed when I look back at the ten years at Microsoft Research Silicon Valley (MSR SVC) during which I had the pleasure of working with and alongside Leslie.

For me—despite him having strong opinions and strong wills, and the fact that when you work with him you always feel inferior because he's so good—I love working with him. Whenever he had the availability to work with me, he was my top choice, no question. I don't say this vacuously: There are people who are difficult to work with, and you don't care how smart or good they are, you just don't want to work with them—it's not worth it. That's not Leslie. Everything he argues about comes from a completely technical point of view—there's nothing personal.

I believe that his personal influence on people in the MSR SVC lab and beyond is priceless. Many of the SVC researchers whom I talked to received life advice and wisdom from him beyond whatever technical matters they discussed with him. They really appreciated him as a role model and got advice from him on how to think about a problem, what to look for in the problems that you work on, and how to think about your career as an industrial researcher. He used to tell us: "Go to engineers and see what problems they are tackling—this is the best source of questions." His confidence is something that I think rubbed off on all of us, that sticking to our inner (scientific) truth and was more important than getting papers published. Beyond the lab, his personal web page and all the stories about rejected papers are an inspiration and an encouragement to every graduate student.

I was at Lamport's Turing Award lecture at PODC in France. He came on stage, was introduced, and got, as you would expect, a standing ovation. Unexpectedly, the ovation did not last 10 seconds—it lasted a solid few minutes. The audience just didn't stop—completely spontaneously. He was stunned and really humbled that people loved him that much.

Roy Levin

I never collaborated with Lamport on a technical matter, though during my years as a researcher I did once seek his guidance on a formal specification for a complex system. However, as the director of SRC and subsequently as director of Microsoft Research Silicon Valley, I frequently sought his wisdom on hiring researchers. Everyone in the lab participated in hiring decisions, and we shared a view of what qualities were essential in our researchers. Lamport regularly reminded us of the importance of focusing on those qualities when considering a candidate. In our lab-wide hiring discussions, Lamport often did not feel the need to speak up, but when he did (as in the old E. F. Hutton ads), people listened. Lamport interviewed many candidates whose expertise didn't overlap with his own; nevertheless, he accurately

assessed their overall technical strength, and when the lab found itself uncertain about the correct decision, he could bring the question into sharp focus. Years later, I can clearly remember several cases in which he made the key characterization of a candidate that cleared away the clouds obscuring the correct hiring decision. We considered that choosing our colleagues was the most important thing the lab did, and Lamport's incisive contributions to that process were therefore invaluable.

It is appropriate in a biography to let the subject have the last words. Here are Lamport's [[Lamport and Levin 2016b](#), page 31]:

One effect that winning the [Turing] award had on me is that it got me to look back at my career in ways that I hadn't, and I think it made me realize the debt that I owed to other computer scientists that I hadn't realized before. For example, when I look back at Dijkstra's mutual exclusion paper now—I've recognized for decades what an amazing paper that was in terms of the insight that he showed, both in recognizing the problem and in stating it so precisely and so accurately—one thing that I didn't realize, I think, until fairly recently, is how much of the whole way of looking at the idea of proving something about an algorithm was new in that paper. He was assuming an underlying model of computation that I somehow accepted as being quite natural. I don't think I understood until recently how much he created that, and I think there are some other instances like that where I absorbed things from other people without realizing it. One of the reasons for that may be that I never had a computer science mentor. . . . I never got that in school really—I never had a one-on-one relationship with any professor. So, the whole concept of mentoring is somewhat alien to me. I hope that, in the couple of occasions where I've been in a position to mentor others, I didn't do too bad a job, but I have no idea whether I did or not. But my mentors have been my colleagues and I learned a lot from them by osmosis that, in retrospect, I'm very grateful for, but I was unaware at the time.



Leslie Lampert and wife Ellen Gilkerson near Monterey, CA, January 24, 2003.



Leslie Lampert and friends at a winery in Napa or Sonoma in 2001. Left to right: Lampert, Ellen Gilkerson, Mimi Bussan, Fred Schneider, Susan Armstrong. (Photo by Keith Marzullo)

PART

SELECTED PAPERS

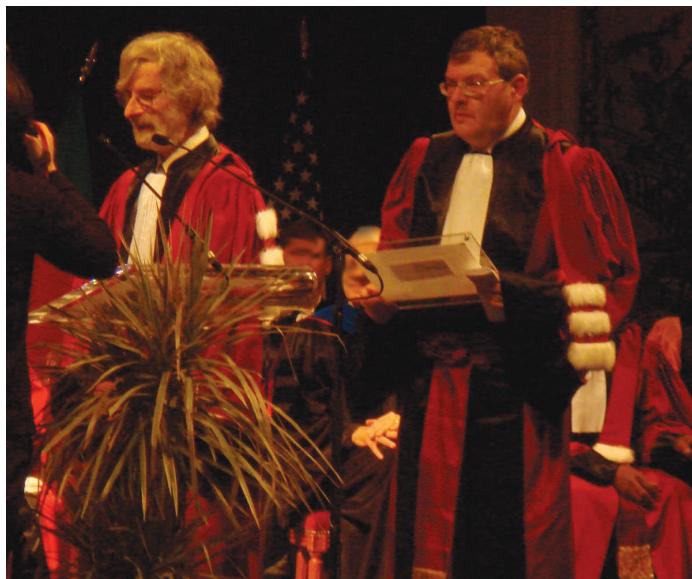


Richard Palais and Leslie Lamport at a lunch in celebration of Palais's 80th birthday in 2011.

Leslie says: "Its significance is that he was my de jure thesis adviser and was the most influential post-high school teacher I had. (It's impossible for me to compare his influence with that of earlier teachers who taught me at a more impressionable age.) He showed me that real math could be made completely rigorous."



Leslie Lamport receives his first honorary doctorate in Rennes, France, in 2003.



Leslie Lamport receives an honorary degree in Nancy. He is standing next to Dominique Mery.

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport (Massachusetts Computer Associates, Inc.)

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate despite the failure of any individual component.

Key Words and Phrases: critical section, concurrent programming, multiprocessing, semaphores

CR Categories: 4.32

Introduction

Knuth [1], deBruijn [2], and Eisenberg and McGuire [3] have given solutions to a concurrent programming problem originally proposed and solved by Dijkstra [4]. A simpler solution using semaphores has also been implemented [5]. These solutions have one drawback for use in a true multicomputer system (rather than a time-shared multiprocessor system): the failure of a single unit will halt the entire system. We present a simple solution which allows the system to continue to operate despite the failure of any individual component.

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by U.S. Army Research Office-Durham, under Contract No. DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, MA 01880.

Paper originally published in *Communications of the ACM* 17(8), August 1974.

The Algorithm

Consider N asynchronous computers communicating with each other only via shared memory. Each computer runs a cyclic program with two parts—a *critical section* and a *noncritical section*. Dijkstra's problem, as extended by Knuth, is to write the programs so that the following conditions are satisfied:

1. At any time, at most one computer may be in its critical section.
2. Each computer must eventually be able to enter its critical section (unless it halts).
3. Any computer may halt in its noncritical section.

Moreover, no assumptions can be made about the running speeds of the computers.

The solutions of [1–4] had all N processors set and test the value of a single variable k . Failure of the memory unit containing k would halt the system. The use of semaphores also implies reliance upon a single hardware component.

Our solution assumes N processors, each containing its own memory unit. A processor may read from any other processor's memory, but it need only write into its own memory. The algorithm has the remarkable property that if a read and a write operation to a single memory location occur simultaneously, then only the write operation must be performed correctly. The read may return *any* arbitrary value!

A processor may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts. There may then be a period when reading from its memory gives arbitrary values. Eventually, any read from its memory must give a value of zero. (In practice, a failed computer might be detected by its failure to respond to a read request within a specified length of time.)

Unlike the solutions of [1–4], ours is a first-come-first-served method in the following sense. When a processor wants to enter its critical section, it first executes a loop-free block of code—i.e. one with a fixed number of execution steps. It is then guaranteed to enter its critical section before any other processor which later requests service.

The algorithm is quite simple. It is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store. The holder of the lowest number is the next one served. In our algorithm, each processor chooses its own number. The processors are named $1, \dots, N$. If two processors choose the same number, then the one with the lowest name goes first.

The common store consists of

integer array *choosing*[1 : *N*], *number*[1 : *N*]

Words *choosing*(*i*) and *number*[*i*] are in the memory of processor *i*, and are initially zero. The range of value of *number*[*i*] is unbounded. This will be discussed below.

The following is the program for processor *i*. Execution must begin inside the noncritical section. The arguments of the maximum function can be read in any order. The relation “less than” on ordered pairs of integers is defined by $(a, b) < (c, d)$ if $a < c$, or if $a = c$ and $b < d$.

```

begin integer j;
    L1:   choosing[i]:= 1;
            number[i]:= 1 + maximum(number[1], . . . , number[N]);
            choosing[i]:= 0;
            for j = 1 step 1 until N do
                begin
                    L2:   if choosing[j] ≠ 0 then goto L2;
                    L3:   if number[j] ≠ 0 and (number[j], j) < (number[i], i)
                            then goto L3;
                end;
                critical section;
                number[i]:= 0;
                noncritical section;
                goto L1;
end

```

We allow process *i* to fail at any time, and then to be restarted in its noncritical sections (with *choosing*[*i*] = *number*[*i*] = 0). However, if a processor keeps failing and restarting, then it can deadlock the system.

Proof of Correctness

To prove the correctness of the algorithm, we first make the following definitions. Processor *i* is said to be *in the doorway* while *choosing*[*i*] = 1. It is said to be *in the bakery* from the time it resets *choosing*(*i*) to zero until it either fails or leaves its critical section. The correctness of the algorithm is deduced from the following assertions. Note that the proofs make no assumptions about the value read during an overlapping read and write to the same memory location.

Assertion 1 If processor i and k are in the bakery and i entered the bakery before k entered the doorway, then $number[i] < number[k]$.

Proof By hypothesis, $number[i]$ had its current value while k was choosing the current value of $number[k]$. Hence, k must have chosen $number \geq 1 + number[i]$. ■

Assertion 2 If processor i is in its critical section, processor k is in the bakery, and $k \neq i$, then $(number[i], i) < (number[k], k)$.

Proof Since $choosing[k]$ has essentially just two values—zero and nonzero—we can assume that from processor i 's point of view, reading or writing it is done instantaneously, and a simultaneous read and write does not occur. For example, if $choosing[k]$ is being changed from zero to one while it is also being read by processor i , then the read is considered to happen first if it obtains a value of zero; otherwise the write is said to happen first. All times defined in the proof are from processor i 's viewpoint.

Let t_{L2} be the time at which processor i read $choosing[k]$ during its last execution of $L2$ for $j = k$, and let t_{L3} be the time at which i began its last execution of $L3$ for $j = k$, so $t_{L2} < t_{L3}$. When processor k was choosing its current value of $number[k]$, let t_e be the time at which it entered the doorway, t_w the time at which it finished writing the value of $number[k]$, and t_c the time at which it left the doorway. Then $t_e < t_w < t_c$.

Since $choosing[k]$ was equal to zero at time t_{L2} , we have either (a) $t_{L2} < t_e$ or (b) $t_c < t_{L2}$. In case (a), Assertion 1 implies that $number[i] < number[k]$, so the assertion holds.

In case (b), we have $t_w < t_c < t_{L2} < t_{L3}$, so $t_w < t_{L3}$. Hence, during the execution of statement $L3$ begun at time t_{L3} , processor i read the current value of $number[k]$. Since i did not execute $L3$ again for $j = k$, it must have found $(number[i], i) < (number[k], k)$. Hence, the assertion holds in this case, too. ■

Assertion 3 Assume that only a bounded number of processor failures may occur. If no processor is in its critical section and there is a processor in the bakery which does not fail, then some processor must eventually enter its critical section.

Proof Assume that no processor ever enters its critical section. Then there will be some time after which no more processors enter or leave the bakery. At this time, assume that processor i has the minimum value of $(number[i], i)$ among all processors in the bakery. Then processor i must eventually complete the **for** loop and enter its critical section. This is the required contradiction. ■

Assertion 2 implies that at most one processor can be in its critical section at any time. Assertions 1 and 2 prove that processors enter their critical sections on a first-come-first-served basis. Hence, an individual processor cannot be blocked unless the entire system is deadlocked. Assertion 3 implies that the system can only be deadlocked by a processor halting in its critical section, or by an unbounded sequence of processor failures and re-entries. The latter can tie up the system as follows. If processor j continually fails and restarts, then with bad luck processor i could always find $choosing[j] = 1$, and loop forever at $L2$.

Further Remarks

If there is always at least one processor in the bakery, then the value of $number[i]$ can become arbitrarily large. This problem cannot be solved by any simple scheme of cycling through a finite set of integers. For example, given any numbers r and s , if $N \geq 4$, then it is possible to have simultaneously $number(i) = r$ and $number(j) = s$ for some i and j .

Fortunately, practical considerations will place an upper bound on the value of $number[i]$ in any real application. For example, if processors enter the doorway at the rate of at most one per msec, then after a year of operation we will have $number[i] < 2^{35}$ —assuming that a read of $number[i]$ can never obtain a value larger than one which has been written there.

The unboundedness of $number[i]$ does raise an interesting theoretical question: can one find an algorithm for finite processors such that processors enter their critical sections on a first-come-first-served basis, and no processor may write into another processor's memory? The answer is not known.¹

The algorithm can be generalized in two ways: (i) under certain circumstances, to allow two processors simultaneously to be in their critical sections; and (ii) to modify the first-come-first-served property so that higher priority processors are served first. This will be described in a future paper.

Conclusion

Our algorithm provides a new, simple solution of the mutual exclusion problem. Since it does not depend upon any form of central control, it is less sensitive to component failure than previous solutions.

Received September 1973; revised January 1974

1. We have recently found such an algorithm, but it is quite complicated.

References

- [1] Knuth, D.E. Additional comments on a problem in concurrent programming control. *Comm. ACM* 9, 5 (May 1966), 321–322.
- [2] deBruijn, N.G. Additional comments on a problem in concurrent programming control. *Comm. ACM* 10, 3 (Mar. 1967), 137–138.
- [3] Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. *Comm. ACM* 15, 11 (Nov. 1972), 999.
- [4] Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (Sept. 1965), 569.
- [5] Dijkstra, E.W. The structure of THE multiprogramming system. *Comm. ACM* 11, 5 (May 1968), 341–346.

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport (Massachusetts Computer Associates, Inc.)

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

© 1978 ACM 0001-0782/78/0700-0558 \$00.75

Paper originally published in *Communications of the ACM* 21(7), July 1978.

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and before it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation “happened before” is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the “happened before” relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of

physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the “happened before” relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one event. We are assuming that the events of a process form a sequence, where a occurs before b in this sequence if a happens before b . In other words, a single process is defined to be a set of events with an *a priori* total ordering. This seems to be what is generally meant by a process.¹ It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the “happened before” relation, denoted by “ \rightarrow ”, as follows.

Definition

The relation “ \rightarrow ” on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b , then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

We assume that $a \not\rightarrow a$ for any event a . (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that \rightarrow is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a “space-time diagram” such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.² It is easy

-
1. The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process’ message-receiving events.
 2. Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

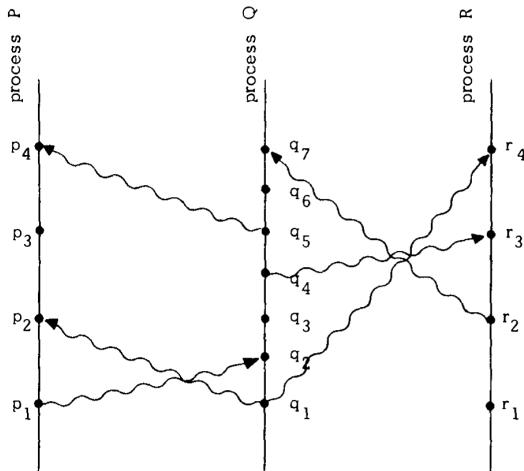


Figure 1

to see that $a \rightarrow b$ means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure 1.

Another way of viewing the definition is to say that $a \rightarrow b$ means that it is possible for event a to causally affect event b . Two events are concurrent if neither can causally affect the other. For example, events p_3 and q_3 of Figure 1 are concurrent. Even though we have drawn the diagram to imply that q_3 occurs at an earlier physical time than p_3 , process P cannot know what process Q did at q_3 until it receives the message at p_4 . (Before event p_4 , P could at most know what Q was *planning* to do at q_3 .)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a

clock C_i for each process P_i to be a function which assigns a number $C_i(a)$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in process P_j . For now, we make no assumption about the relation of the numbers $C_i(a)$ to physical time, so we can think of the clocks C_i as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event a occurs before another event b , then a should happen at an earlier time than b . We state this condition more formally as follows.

Clock Condition For any events a, b : if $a \rightarrow b$ then $C(a) < C(b)$.

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1, p_2 and p_3 are both concurrent with q_3 , so this would mean that they both must occur at the same time as q_3 , which would contradict the Clock Condition because $p_2 \rightarrow p_3$.

It is easy to see from our definition of the relation “ \rightarrow ” that the Clock Condition is satisfied if the following two conditions hold.

- C1** If a and b are events in process P_i , and a comes before b , then $C_i(a) < C_i(b)$.
- C2** If a is the sending of a message by process P_i and b is the receipt of that message by process P_j , then $C_i(a) < C_j(b)$.

Let us consider the clocks in terms of a space-time diagram. We imagine that a process’ clock “ticks” through every number, with the ticks occurring between the process’ events. For example, if a and b are consecutive events in process P_i with $C_i(a) = 4$ and $C_i(b) = 7$, then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed “tick line” through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and condition C2 means that every message line must cross a tick line. From the pictorial meaning of \rightarrow , it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these

184 Time, Clocks, and the Ordering of Events in a Distributed System

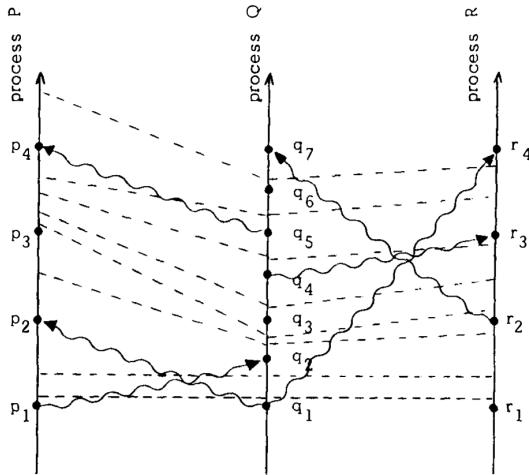


Figure 2

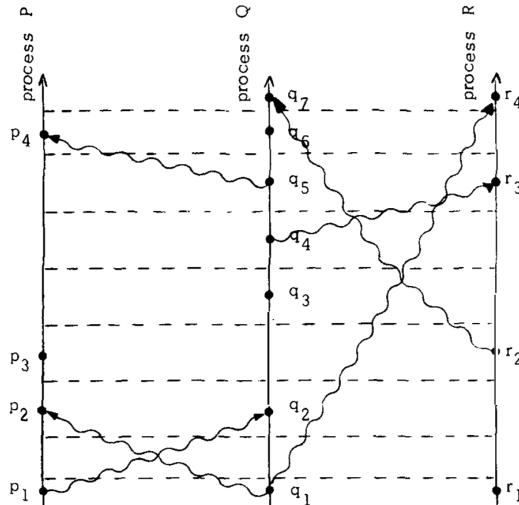


Figure 3

coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process P_i 's clock is represented by a register C_i , so that $C_i(a)$ is the value contained by C_i during the event a . The value of C_i will change between events, so changing C_i does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

- IR1** Each process P_i increments C_i between any two successive events.

To meet condition C2, we require that each message m contain a *timestamp* T_m which equals the time at which the message was sent. Upon receiving a message timestamped T_m , a process must advance its clock to be later than T_m . More precisely, we have the following rule.

- IR2** (a) If event a is the sending of a message m by process P_i , then the message m contains a timestamp $T_m = C_i(a)$. (b) Upon receiving a message m , process P_j sets C_j greater than or equal to its present value and greater than T_m .

In IR2(b) we consider the event which represents the receipt of the message m to occur after the setting of C_j . (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times at which they occur. To break ties, we use any arbitrary total ordering \prec of the processes. More precisely, we define a relation \Rightarrow as follows: if a is an event in process P_i and b is an event in process P_j , then $a \Rightarrow b$ if and only if either (i) $C_i(a) < C_j(b)$ or (ii) $C_i(a) = C_j(b)$ and $P_i \prec P_j$. It is easy to see that this defines a total ordering, and that the Clock Condition implies that if $a \rightarrow b$ then $a \Rightarrow b$. In

other words, the relation \Rightarrow is a way of completing the “happened before” partial ordering to a total ordering.³

The ordering \Rightarrow depends upon the system of clocks C_i , and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations \Rightarrow . Given any total ordering relation \Rightarrow which extends \rightarrow , there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering \rightarrow which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.⁴ Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let P_0 be the scheduling process. Suppose P_1 sends a request to P_0 and then sends a message to P_2 . Upon receiving the latter message, P_2 sends a request to P_0 . It is possible for P_2 ’s request to reach P_0 before P_1 ’s request does. Condition II is then violated if P_2 ’s request is granted first.

To solve the problem, we implement a system of clocks with rules IR1 and IR2, and use them to define a total ordering \Rightarrow of all events. This provides a total

3. The ordering \prec establishes a priority among the processes. If a “fairer” method is desired, then \prec can be made a function of the clock value. For example, if $C_i(a) = C_j(b)$ and $j < i$, then we can let $a \Rightarrow b$ if $j < C_i(a) \bmod N \leq i$, and $b \Rightarrow a$ otherwise; where N is the total number of processes.

4. The term “eventually” should be made precise, but that would require too long a diversion from our main topic.

ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes' operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes P_i and P_j , the messages sent from P_i to P_j are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message $T_0: P_0 \text{ requests resource}$, where P_0 is the process initially granted the resource and T_0 is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message $T_m: P_i \text{ requests resource}$ to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.
2. When process P_j receives the message $T_m: P_i \text{ requests resource}$, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i .⁵
3. To release the resource, process P_i removes any $T_m: P_i \text{ requests resource}$ message from its request queue and sends a (timestamped) $P_i \text{ releases resource}$ message to every other process.
4. When process P_j receives a $P_i \text{ releases resource}$ message, it removes any $T_m: P_i \text{ requests resource}$ message from its request queue.
5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a $T_m: P_i \text{ requests resource}$ message in its request queue which is ordered before any other request in its queue by the relation \Rightarrow . (To define the relation " \Rightarrow " for messages, we identify a message with the event of sending it.) (ii) P_i has received a message from every other process timestamped later than T_m .⁶

5. This acknowledgment message need not be sent if P_j has already sent a message to P_i timestamped later than T_m .

6. If $P_i \prec P_j$, then P_i need only have received a message timestamped $\geq T_m$ from P_j .

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i .

It is easy to verify that the algorithm defined by these rules satisfies conditions I–III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that P_i has learned about all requests which preceded its current request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering \Rightarrow extends the partial ordering \rightarrow . Rule 2 guarantees that after P_i requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set C of possible commands, a set S of possible states, and a function $e: C \times S \rightarrow S$. The relation $e(C, S) = S'$ means that executing the command C with the machine in state S causes the machine state to change to S' . In our example, the set C consists of all the commands P_i *requests resource* and P_i *releases resource*, and the state consists of a queue of waiting request commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.⁷

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation \Rightarrow), so each process uses the same sequence of commands. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T . The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impos-

7. If each process does not strictly alternate request and release commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

sible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has “crashed” only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering \Rightarrow . This permits the following type of “anomalous behavior.” Consider a nationwide system of interconnected computers. Suppose a person issues a request A on a computer A , and then telephones a friend in another city to have him issue a request B on a different computer B . It is quite possible for request B to receive a lower timestamp and be ordered before request A . This can happen because the system has no way of knowing that A actually preceded B , since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let \mathcal{S} be the set of all system events. Let us introduce a set of events which contains the events in \mathcal{S} together with all other relevant external events, such as the phone calls in our example. Let \rightarrow denote the “happened before” relation for \mathcal{S} . In our example, we had $A \rightarrow B$, but $A \not\rightarrow B$. It is obvious that no algorithm based entirely upon events in \mathcal{S} , and which does not relate those events in any way with the other events in \mathcal{S} , can guarantee that request A is ordered before request B .

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering \rightarrow . In our example, the person issuing request A could receive the timestamp T_A of that request from the system. When issuing request B , his friend could specify that B be given a timestamp later than T_A . This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

Strong Clock Condition For any events a, b in \mathcal{S} : if $a \rightarrow b$ then $C(a) < C(b)$.

This is stronger than the ordinary Clock Condition because \rightarrow is a stronger relation than \rightarrow . It is not in general satisfied by our logical clocks.

Let us identify \mathcal{S} with some set of “real” events in physical space-time, and let \rightarrow be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let $C_i(t)$ denote the reading of the clock C_i at physical time t .⁸ For mathematical convenience, we assume that the clocks run continuously rather than in discrete “ticks.” (A discrete clock can be thought of as a continuous one in which there is an error of up to $1/2$ “tick” in reading it.) More precisely, we assume that $C_i(t)$ is a continuous, differentiable function of t except for isolated jump discontinuities where the clock is reset. Then $dC_i(t)/dt$ represents the rate at which the clock is running at time t .

In order for the clock C_i to be a true physical clock, it must run at approximately the correct rate. That is, we must have $dC_i(t)/dt \approx 1$ for all t . More precisely, we will assume that the following condition is satisfied:

- PC1** There exists a constant $\kappa \ll 1$ such that for all i : $|dC_i(t)/dt - 1| < \kappa$.

For typical crystal controlled clocks, $\kappa \leq 10^{-6}$.

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that $C_i(t) \approx C_j(t)$ for all i , j , and t . More precisely, there must be a sufficiently small constant ϵ so that the following condition holds:

- PC2** For all i , j : $|C_i(t) - C_j(t)| < \epsilon$.

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than ϵ .

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure

8. We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then $C_i(t)$ must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

that PC2 always holds. First, however, let us examine how small κ and ϵ must be to prevent anomalous behavior. We must insure that the system \mathcal{S} of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when a and b are events in \mathcal{S} with $a \not\rightarrow b$. Hence, we need only consider events occurring in different processes.

Let μ be a number such that if event a occurs at physical time t and event b in another process satisfies $a \rightarrow b$, then b occurs later than physical time $t + \mu$. In other words, μ is less than the shortest transmission time for interprocess messages. We can always choose μ equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in \mathcal{S} are transmitted, μ could be significantly larger.

To avoid anomalous behavior, we must make sure that for any i, j , and t : $C_i(t + \mu) - C_j(t) > 0$. Combining this with PC1 and 2 allows us to relate the required smallness of κ and ϵ to the value of μ as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$. Using PC2, it is then easy to deduce that $C_i(t + \mu) - C_j(t) > 0$ if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let m be a message which is sent at physical time t and received at time t' . We define $\nu_m = t' - t$ to be the *total delay* of the message m . This delay will, of course, not be known to the process which receives m . However, we assume that the receiving process knows some *minimum delay* $\mu_m \geq 0$ such that $\mu_m \leq \nu_m$. We call $\xi_m = \nu_m - \mu_m$ the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

- IR1**: For each i , if P_i does not receive a message at physical time t , then C_i is differentiable at t and $dC_i(t)/dt > 0$.
- IR2**: (a) If P_i sends a message m at physical time t , then m contains a timestamp $T_m = C_i(t)$. (b) Upon receiving a message m at time t' , process P_j sets $C_j(t')$ equal to $\max(C_j(t' - 0), T_m + \mu_m)$.⁹

9. $C_j(t' - 0) = \lim_{\delta \rightarrow 0} C_j(t' - |\delta|)$.

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process P_i to process P_j represents a communication line over which messages are sent directly from P_i to P_j . We say that a message is sent over this arc every τ seconds if for any t , P_i sends at least one message to P_j between physical times t and $t + \tau$. The *diameter* of the directed graph is the smallest number d such that for any pair of distinct processes P_j, P_k , there is a path from P_j to P_k having at most d arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

Theorem Assume a strongly connected graph of processes with diameter d which always obeys rules IR1' and IR2'. Assume that for any message m , $\mu_m \leq \mu$ for some constant μ , and that for all $t \geq t_0$: (a) PC1 holds. (b) There are constants τ and ξ such that every τ seconds a message with an unpredictable delay less than ξ is sent over every arc. Then PC2 is satisfied with $\epsilon \approx d(2\kappa\tau + \xi)$ for all $t \geq t_0 + \tau d$, where the approximations assume $\mu + \xi \ll \tau$.

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an introduction to the subject. The methods described in the literature are useful for estimating the message delays μ_m and for adjusting the clock frequencies dC_i/dt (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

Conclusion

We have seen that the concept of “happening before” defines an invariant partial ordering of the events in a distributed multiprocess system. We described an al-

gorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system's users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

Appendix

Proof of the Theorem

For any i and t , let us define C_i^t to be a clock which is set equal to C_i at time t and runs at the same rate as C_i , but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt] dt \quad (1)$$

for all $t' \geq t$. Note that

$$C_i(t') \geq C_i^t(t') \quad \text{for all } t' \geq t. \quad (2)$$

Suppose process P_1 at time t_1 sends a message to process P_2 which is received at time t_2 with an unpredictable delay $\leq \xi$, where $t_0 \leq t_1 \leq t_2$. Then for all $t \geq t_2$ we have:

$$\begin{aligned} C_2^{t_2}(t) &\geq C_2^{t_2}(t_2) + (1 - \kappa)(t - t_2) && [\text{by (1) and PC1}] \\ &\geq C_1(t_1) + \mu_m + (1 - \kappa)(t - t_2) && [\text{by IR2'(b)}] \\ &= C_1(t_1) + (1 - \kappa)(t - t_1) - [(t_2 - t_1) - \mu_m] + \kappa(t_2 - t_1) \\ &\geq C_1(t_1) + (1 - \kappa)(t - t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all $t \geq t_2$ we have:

$$C_2^{t_2}(t) \geq C_1(t_1) + (1 - \kappa)(t - t_1) - \xi. \quad (3)$$

Now suppose that for $i = 1, \dots, n$ we have $t_i \leq t'_i < t_{i+1}$, $t_0 \leq t_1$, and that at time t'_i process P_i sends a message to process P_{i+1} which is received at time t_{i+1} with an unpredictable delay less than ξ . Then repeated application of the inequality (3)

yields the following result for $t \geq t_{n+1}$:

$$C_{n+1}^{t_{n+1}}(t) \geq C_1(t'_1) + (1 - \kappa)(t - t'_1) - n\xi. \quad (4)$$

From PC1, IR1' and 2' we deduce that

$$C_1(t'_1) \geq C_1(t_1) + (1 - \kappa)(t'_1 - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \geq C_1(t_1) + (1 - \kappa)(t - t_1) - n\xi \quad (5)$$

for $t \geq t_{n+1}$.

For any two processes P and P' , we can find a sequence of processes $P = P_0, P_1, \dots, P_{n+1} = P'$, $n \leq d$, with communication arcs from each P_i to P_{i+1} . By hypothesis (b) we can find times t_i, t'_i with $t'_i - t_i \leq \tau$ and $t_{i+1} - t'_i \leq \nu$, where $\nu = \mu + \xi$. Hence, an inequality of the form (5) holds with $n \leq d$ whenever $t \geq t_1 + d(\tau + \nu)$. For any i, j and any t, t_1 with $t_1 \geq t_0$ and $t \geq t_1 + d(\tau + \nu)$ we therefore have:

$$C_i(t) \geq C_j(t_1) + (1 - \kappa)(t - t_1) - d\xi. \quad (6)$$

Now let m be any message timestamped T_m , and suppose it is sent at time t and received at time t' . We pretend that m has a clock C_m which runs at a constant rate such that $C_m(t) = t_m$ and $C_m(t') = t_m + \mu_m$. Then $\mu_m \leq t' - t$ implies that $dC_m/dt \leq 1$. Rule IR2'(b) simply sets $C_j(t')$ to $\max(C_j(t' - 0), C_m(t'))$. Hence, clocks are reset only by setting them equal to other clocks.

For any time $t_x \geq t_0 + \mu/(1 - \kappa)$, let C_x be the clock having the largest value at time t_x . Since all clocks run at a rate less than $1 + \kappa$, we have for all i and all $t \geq t_x$:

$$C_i(t) \leq C_x(t_x) + (1 + \kappa)(t - t_x). \quad (7)$$

We now consider the following two cases: (i) C_x is the clock C_q of process P_q . (ii) C_x is the clock C_m of a message sent at time t_1 by process P_q . In case (i), (7) simply becomes

$$C_i(t) \leq C_q(t_x) + (1 + \kappa)(t - t_x). \quad (8i)$$

In case (ii), since $C_m(t_1) = C_q(t_1)$ and $dC_m/dt \leq 1$, we have

$$C_x(t_x) \leq C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \quad (8ii)$$

Since $t_x \geq t_0 + \mu/(1 - \kappa)$, we get

$$\begin{aligned}
 C_q(t_x - \mu/(1 - \kappa)) &\leq C_q(t_x) - \mu && [\text{by PC1}] \\
 &\leq C_m(t_x) - \mu && [\text{by choice of } m] \\
 &\leq C_m(t_x) - (t_x - t_1)\mu_m/v_m && [\mu_m \leq \mu, t_x - t_1 \leq v_m] \\
 &= T_m && [\text{by definition of } C_m] \\
 &= C_q(t_1) && [\text{by IR2'(a)}]
 \end{aligned}$$

Hence, $C_q(t_x - \mu/(1 - \kappa)) \leq C_q(t_1)$, so $t_x - t_1 \leq \mu/(1 - \kappa)$ and thus $t_1 \geq t_0$.

Letting $t_1 = t_x$ in case (i), we can combine (8i) and (8ii) to deduce that for any t, t_x , with $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$ there is a process P_q and a time t_1 with $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$ such that for all i :

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \quad (9)$$

Choosing t and t_x with $t \geq t_x + d(\tau + v)$, we can combine (6) and (9) to conclude that there exists a t_1 and a process P_q such that for all i :

$$\begin{aligned}
 C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi &\leq C_i(t) \\
 &\leq C_q(t_1) + (1 + \kappa)(t - t_1)
 \end{aligned} \quad (10)$$

Letting $t = t_x + d(\tau + v)$, we get

$$d(\tau + v) \leq t - t_1 \leq d(\tau + v) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$\begin{aligned}
 C_q(t_1) + (t - t_1) - \kappa d(\tau + v) - d\xi &\leq C_i(t)C_q(t_1) \\
 &\quad + (t - t_1) + \kappa[d(\tau + v) + \mu/(1 - \kappa)]
 \end{aligned} \quad (11)$$

Using the hypothesis that $\kappa \ll 1$ and $\mu \leq v \ll \tau$, we can rewrite (11) as the following approximate inequality.

$$\begin{aligned}
 C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) &\lesssim C_i(t) \\
 &\lesssim C_q(t_1) + (t - t_1) + d\kappa\tau.
 \end{aligned} \quad (12)$$

Since this holds for all i , we get

$$|C_i(t) - C_j(t)| \lesssim d(2\kappa\tau + \xi),$$

and this holds for all $t \gtrsim t_0 + dt$. ■

Note that relation (11) of the proof yields an exact upper bound for $|C_i(t) - C_j(t)|$ in case the assumption $\mu + \xi \ll \tau$ is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than $2d(\mu + \xi)$ seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than ξ .

Acknowledgment. The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

References

- [1] Schwartz, J.T. *Relativity in Illustrations*. New York U. Press, New York, 1962.
- [2] Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
- [3] Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks*.
- [4] Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605-624.

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

Leslie Lamport

Abstract—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

Index Terms—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for

Manuscript received September 28, 1977; revised May 8, 1979.

The author is with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025. Paper originally published in *IEEE Transactions on Computers* C-28(9), September 1979, pp. 690–691.

such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

We assume that the computer consists of a collection of processors and memory modules, and that the processors communicate with one another only through the memory modules. (Any special communication registers may be regarded as separate memory modules.) The only processor operations that concern us are the operations of sending fetch and store requests to memory modules. We assume that each processor issues a sequence of such requests. (It must sometimes wait for requests to be executed, but that does not concern us.)

We illustrate the problem by considering a simple two-process mutual exclusion protocol. Each process contains a *critical section*, and the purpose of the protocol is to insure that only one process may be executing its critical section at any time. The protocol is as follows.

```

process 1
  a := 1;
  if b = 0 then critical section;
    a := 0
  else . . . fi

process 2
  b := 1;
  if a = 0 then critical section;
    b := 0
  else . . . fi

```

The else clauses contain some mechanism for guaranteeing eventual access to the critical section, but that is irrelevant to the discussion. It is easy to prove that this protocol guarantees mutually exclusive access to the critical sections. (Devising a proof provides a nice exercise in using the assertional techniques of [2] and [3], and is left to the reader.) Hence, when this two-process program is executed by a sequentially consistent multiprocessor computer, the two processors cannot both be executing their critical sections at the same time.

We first observe that a sequential processor could execute the “ $b := 1$ ” and “fetch b ” operations of process 1 in either order. (When process 1’s program is considered by itself, it does not matter in which order these two operations are performed.) However, it is easy to see that executing the “fetch b ” operation first can lead to an error—both processes could then execute their critical sections at the same time. This immediately suggests our first requirement for a multiprocessor computer.

Requirement R1 Each processor issues memory requests in the order specified by its program.

Satisfying Requirement R1 is complicated by the fact that storing a value is possible only after the value has been computed. A processor will often be ready to issue a memory fetch request before it knows the value to be stored by a preceding store request. To minimize waiting, the processor can issue the store request to the memory module without specifying the value to be stored. Of course, the store request cannot actually be executed by the memory module until it receives the value to be stored.

Requirement R1 is not sufficient to guarantee correct execution. To see this, suppose that each memory module has several ports, and each port services one processor (or I/O channel). Let the values of “ a ” and “ b ” be stored in separate memory modules, and consider the following sequence of events.

1. Processor 1 sends the “ $a := 1$ ” request to its port in memory module 1. The module is currently busy executing an operation for some other processor (or I/O channel).
2. Processor 1 sends the “fetch b ” request to its port in memory module 2. The module is free, and execution is begun.
3. Processor 2 sends its “ $b := 1$ ” request to memory module 2. This request will be executed after processor 1’s “fetch b ” request is completed.
4. Processor 2 sends its “fetch a ” request to its port in memory module 1. The module is still busy.

There are now two operations waiting to be performed by memory module 1. If processor 2’s “fetch a ” operation is performed first, then both processes can enter their critical sections at the same time, and the protocol fails. This could happen if the memory module uses a round robin scheduling discipline in servicing its ports.

In this situation, an error occurs only if the two requests to memory module 1 are not executed in the same order in which they were received. This suggests the following requirement.

Requirement R2 Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Condition R1 implies that a processor may not issue any further memory requests until after its current request has been entered on the queue. Hence, it must wait if the queue is full. If two or more processors are trying to enter requests in the queue at the same time, then it does not matter in which order they are serviced.

Note If a fetch requests the contents of a memory location for which there is already a write request on the queue, then the fetch need not be entered on the queue. It may simply return the value from the last such write request on the queue.

Requirements R1 and R2 insure that if the individual processors are sequential, then the entire multiprocessor computer is sequentially consistent. To demonstrate this, one first introduces a relation \rightarrow on memory requests as follows. Define $A \rightarrow B$ if and only if 1) A and B are issued by the same processor and A is issued before B, or 2) A and B are issued to the same memory module, and A is entered in the queue before B (and is thus executed before B). It is easy to see that R1 and R2 imply that \rightarrow is a partial ordering on the set of memory requests. Using the sequentiality of each processor, one can then prove the following result: each fetch and store operation fetches or stores the same value as if all the operations were executed sequentially in any order such that $A \rightarrow B$ implies that A is executed before B. This in turn proves the sequential consistency of the multiprocessor computer.

Requirement R2 states that a memory module's request queue must be serviced in a FIFO order. This implies that the memory module must remain idle if the request at the head of its queue is a store request for which the value to be stored has not yet been received. Condition R2 can be weakened to allow the memory module to service other requests in this situation. We need only require that all requests to *the same memory cell* be serviced in the order that they appear in the queue. Requests to different memory cells may be serviced out of order. Sequential consistency is preserved because such a service policy is logically equivalent to considering each memory cell to be a separate memory module with its own request queue. (The fact that these modules may share some hardware affects the rate at which they service requests and the capacity of their queues, but it does not affect the logical property of sequential consistency.)

The requirements needed to guarantee consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing

down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

References

- [1] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.
- [2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 125–143, Mar. 1977.
- [3] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. Assoc. Comput. Mach.*, vol. 19, pp. 279–285, May 1976.

The Byzantine Generals Problem

Leslie Lamport (SRI International),
Robert Shostak (SRI International),
Marshall Pease (SRI International)

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of

This research was supported in part by the National Aeronautics and Space Administration under contract NAS1-15428 Mod. 3, the Ballistic Missile Defense Systems Command under contract DASG60-78-C-0046, and the Army Research Office under contract DAAG29-79-C-0102.

Authors' address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0164-0925/82/0700-0382 \$00.75

Paper originally published in *Transactions on Programming Languages and Systems*, 4(3), July 1982, pp. 382-401.

generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [Computer-Communication Networks]: Distributed Systems—*network operating systems*; D.4.4 [Operating Systems]: Communications Management—*network communication*; D.4.5 [Operating Systems]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

1 Introduction

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.

We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that

- A All loyal generals decide upon the same plan of action.

The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition A regardless of what the traitors do.

The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to insure that

- B A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Condition B is hard to formalize, since it requires saying precisely what a bad plan is, and we do not attempt to do so. Instead, we consider how the generals reach a decision. Each general observes the enemy and communicates his observations to the others. Let $v(i)$ be the information communicated by the i th general. Each

general uses some method for combining the values $v(1), \dots, v(n)$ into a single plan of action, where n is the number of generals. Condition A is achieved by having all generals use the same method for combining the information, and Condition B is achieved by using a robust method. For example, if the only decision to be made is whether to attack or retreat, then $v(i)$ can be General i 's opinion of which option is best, and the final decision can be based upon a majority vote among them. A small number of traitors can affect the decision only if the loyal generals were almost equally divided between the two possibilities, in which case neither decision could be called bad.

While this approach may not be the only way to satisfy conditions A and B, it is the only one we know of. It assumes a method by which the generals communicate their values $v(i)$ to one another. The obvious method is for the i th general to send $v(i)$ by messenger to each other general. However, this does not work, because satisfying condition A requires that every loyal general obtain the same values $v(1), \dots, v(n)$, and a traitorous general may send different values to different generals. For condition A to be satisfied, the following must be true:

- 1 Every loyal general must obtain the same information $v(1), \dots, v(n)$.

Condition 1 implies that a general cannot necessarily use a value of $v(i)$ obtained directly from the i th general, since a traitorous i th general may send different values to different generals. This means that unless we are careful, in meeting condition 1 we might introduce the possibility that the generals use a value of $v(i)$ different from the one sent by the i th general—even though the i th general is loyal. We must not allow this to happen if condition B is to be met. For example, we cannot permit a few traitors to cause the loyal generals to base their decision upon the values “retreat”, . . . , “retreat” if every loyal general sent the value “attack”. We therefore have the following requirement for each i :

- 2 If the i th general is loyal, then the value that he sends must be used by every loyal general as the value of $v(i)$.

We can rewrite condition 1 as the condition that for every i (whether or not the i th general is loyal),

- 1' Any two loyal generals use the same value of $v(i)$.

Conditions 1' and 2 are both conditions on the single value sent by the i th general. We can therefore restrict our consideration to the problem of how a single general sends his value to the others. We phrase this in terms of a commanding general sending an order to his lieutenants, obtaining the following problem.

Byzantine Generals A commanding general must send an order to his $n - 1$ lieutenant generals such that

- IC1** All loyal lieutenants obey the same order.
- IC2** If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Conditions IC1 and IC2 are called the interactive consistency conditions. Note that if the commander is loyal, then IC1 follows from IC2. However, the commander need not be loyal.

To solve our original problem, the i th general sends his value of $v(i)$ by using a solution to the Byzantine Generals Problem to send the order “use $v(i)$ as my value,” with the other generals acting as the lieutenants.

2 Impossibility Results

The Byzantine Generals Problem seems deceptively simple. Its difficulty is indicated by the surprising fact that if the generals can send only oral messages, then no solution will work unless more than two-thirds of the generals are loyal. In particular, with only three generals, no solution can work in the presence of a single traitor. An oral message is one whose contents are completely under the control of the sender, so a traitorous sender can transmit any possible message. Such a message corresponds to the type of message that computers normally send to one another. In Section 4 we consider signed, written messages, for which this is not true.

We now show that with oral messages no solution for three generals can handle a single traitor. For simplicity, we consider the case in which the only possible decisions are “attack” or “retreat”. Let us first examine the scenario pictured in Figure 1 in which the commander is loyal and sends an “attack” order, but Lieutenant 2 is a traitor and reports to Lieutenant 1 that he received a “retreat” order. For IC2 to be satisfied, Lieutenant 1 must obey the order to attack.

Now consider another scenario, shown in Figure 2, in which the commander is a traitor and sends an “attack” order to Lieutenant 1 and a “retreat” order to Lieutenant 2. Lieutenant 1 does not know who the traitor is, and he cannot tell what message the commander actually sent to Lieutenant 2. Hence, the scenarios in these two pictures appear exactly the same to Lieutenant 1. If the traitor lies consistently, then there is no way for Lieutenant 1 to distinguish between these two situations, so he must obey the “attack” order in both of them. Hence, whenever Lieutenant 1 receives an “attack” order from the commander, he must obey it.

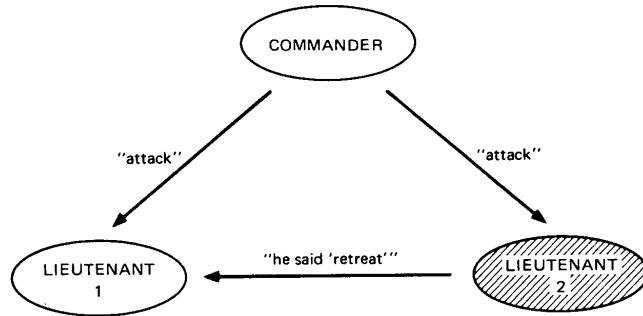


Figure 1 Lieutenant 2 a traitor.

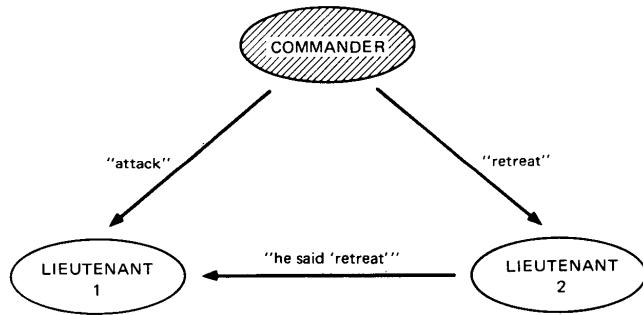


Figure 2 The commander a traitor.

However, a similar argument shows that if Lieutenant 2 receives a “retreat” order from the commander then he must obey it even if Lieutenant 1 tells him that the commander said “attack”. Therefore, in the scenario of Figure 2, Lieutenant 2 must obey the “retreat” order while Lieutenant 1 obeys the “attack” order, thereby violating condition IC1. Hence, no solution exists for three generals that works in the presence of a single traitor.

This argument may appear convincing, but we strongly advise the reader to be very suspicious of such nonrigorous reasoning. Although this result is indeed correct, we have seen equally plausible “proofs” of invalid results. We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm. For a rigorous proof of the impossibility of a three-general solution that can handle a single traitor, we refer the reader to [3].

Using this result, we can show that no solution with fewer than $3m + 1$ generals can cope with m traitors.¹ The proof is by contradiction—we assume such a solution for a group of $3m$ or fewer and use it to construct a three-general solution to the Byzantine Generals Problem that works with one traitor, which we know to be impossible. To avoid confusion between the two algorithms, we call the generals of the assumed solution Albanian generals, and those of the constructed solution Byzantine generals. Thus, starting from an algorithm that allows $3m$ or fewer Albanian generals to cope with m traitors, we construct a solution that allows three Byzantine generals to handle a single traitor.

The three-general solution is obtained by having each of the Byzantine generals simulate approximately one-third of the Albanian generals, so that each Byzantine general is simulating at most m Albanian generals. The Byzantine commander simulates the Albanian commander plus at most $m - 1$ Albanian lieutenants, and each of the two Byzantine lieutenants simulates at most m Albanian lieutenants. Since only one Byzantine general can be a traitor, and he simulates at most m Albanians, at most m of the Albanian generals are traitors. Hence, the assumed solution guarantees that IC1 and IC2 hold for the Albanian generals. By IC1, all the Albanian lieutenants being simulated by a loyal Byzantine lieutenant obey the same order, which is the order he is to obey. It is easy to check that conditions IC1 and IC2 of the Albanian generals solution imply the corresponding conditions for the Byzantine generals, so we have constructed the required impossible solution.

One might think that the difficulty in solving the Byzantine Generals Problem stems from the requirement of reaching exact agreement. We now demonstrate that this is not the case by showing that reaching approximate agreement is just as hard as reaching exact agreement. Let us assume that instead of trying to agree on a precise battle plan, the generals must agree only upon an approximate time of attack. More precisely, we assume that the commander orders the time of the attack, and we require the following two conditions to hold:

- IC1** All loyal lieutenants attack within 10 minutes of one another.
- IC2** If the commanding general is loyal, then every loyal lieutenant attacks within 10 minutes of the time given in the commander's order.

1. More precisely, no such solution exists for three or more generals, since the problem is trivial for two generals.

(We assume that the orders are given and processed the day before the attack and that the time at which an order is received is irrelevant—only the attack time given in the order matters.)

Like the Byzantine Generals Problem, this problem is unsolvable unless more than two-thirds of the generals are loyal. We prove this by first showing that if there were a solution for three generals that coped with one traitor, then we could construct a three-general solution to the Byzantine Generals Problem that also worked in the presence of one traitor. Suppose the commander wishes to send an “attack” or “retreat” order. He orders an attack by sending an attack time of 1:00 and orders a retreat by sending an attack time of 2:00, using the assumed algorithm. Each lieutenant uses the following procedure to obtain his order.

1. After receiving the attack time from the commander, a lieutenant does one of the following:
 - (a) If the time is 1:10 or earlier, then attack.
 - (b) If the time is 1:50 or later, then retreat.
 - (c) Otherwise, continue to step (2).
2. Ask the other lieutenant what decision he reached in step (1).
 - (a) If the other lieutenant reached a decision, then make the same decision he did.
 - (b) Otherwise, retreat.

It follows from IC2' that if the commander is loyal, then a loyal lieutenant will obtain the correct order in step (1), so IC2 is satisfied. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. Since there is at most one traitor, this means that both lieutenants are loyal. It follows from ICI' that if one lieutenant decides to attack in step (1), then the other cannot decide to retreat in step (1). Hence, either they will both come to the same decision in step (1) or at least one of them will defer his decision until step (2). In this case, it is easy to see that they both arrive at the same decision, so IC1 is satisfied. We have therefore constructed a three-general solution to the Byzantine Generals Problem that handles one traitor, which is impossible. Hence, we cannot have a three-general algorithm that maintains ICI' and IC2' in the presence of a traitor.

The method of having one general simulate m others can now be used to prove that no solution with fewer than $3m + 1$ generals can cope with m traitors. The proof is similar to the one for the original Byzantine Generals Problem and is left to the reader.

3 A Solution with Oral Messages

We showed above that for a solution to the Byzantine Generals Problem using oral messages to cope with m traitors, there must be at least $3m + 1$ generals. We now give a solution that works for $3m + 1$ or more generals. However, we first specify exactly what we mean by “oral messages”. Each general is supposed to execute some algorithm that involves sending messages to the other generals, and we assume that a loyal general correctly executes his algorithm. The definition of an oral message is embodied in the following assumptions which we make for the generals’ message system:

- A1** Every message that is sent is delivered correctly.
- A2** The receiver of a message knows who sent it.
- A3** The absence of a message can be detected.

Assumptions A1 and A2 prevent a traitor from interfering with the communication between two other generals, since by A1 he cannot interfere with the messages they do send, and by A2 he cannot confuse their intercourse by introducing spurious messages. Assumption A3 will foil a traitor who tries to prevent a decision by simply not sending messages. The practical implementation of these assumptions is discussed in Section 6.

The algorithms in this section and in the following one require that each general be able to send messages directly to every other general. In Section 5, we describe algorithms which do not have this requirement.

A traitorous commander may decide not to send any order. Since the lieutenants must obey some order, they need some default order to obey in this case. We let RETREAT be this default order.

We inductively define the Oral Message algorithms $OM(m)$, for all nonnegative integers m , by which a commander sends an order to $n - 1$ lieutenants. We show that $OM(m)$ solves the Byzantine Generals Problem for $3m + 1$ or more generals in the presence of at most m traitors. We find it more convenient to describe this algorithm in terms of the lieutenants “obtaining a value” rather than “obeying an order”.

The algorithm assumes a function $majority$ with the property that if a majority of the values v_i equal v , then $majority(v_1, \dots, v_{n-1})$ equals v . (Actually, it assumes a sequence of such functions—one for each n .) There are two natural choices for the value of $majority(v_1, \dots, v_{n-1})$:

1. The majority value among the v_i if it exists, otherwise the value RETREAT;

2. The median of the v_i , assuming that they come from an ordered set.

The following algorithm requires only the aforementioned property of *majority*.

Algorithm OM(0)

1. The commander sends his value to every lieutenant.
2. Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Alg. OM(m), $m > 0$

1. The commander sends his value to every lieutenant.
2. For each i , let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m - 1$) to send the value v_i to each of the $n - 2$ other lieutenants.
3. For each i , and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m - 1$)), or else RETREAT if he received no such value. Lieutenant i uses the value $\text{majority}(v_1, \dots, v_{n-1})$.

To understand how this algorithm works, we consider the case $m = 1, n = 4$. Figure 3 illustrates the messages received by Lieutenant 2 when the commander sends the value v and Lieutenant 3 is a traitor. In the first step of OM(1), the commander sends v to all three lieutenants. In the second step, Lieutenant 1 sends the value v to Lieutenant 2, using the trivial algorithm OM(0). Also in the second step, the traitorous Lieutenant 3 sends Lieutenant 2 some other value x . In step 3, Lieutenant 2 then has $v_1 = v_2 = v$ and $v_3 = x$, so he obtains the correct value $v = \text{majority}(v, v, x)$.

Next, we see what happens if the commander is a traitor. Figure 4 shows the values received by the lieutenants if a traitorous commander sends three arbitrary

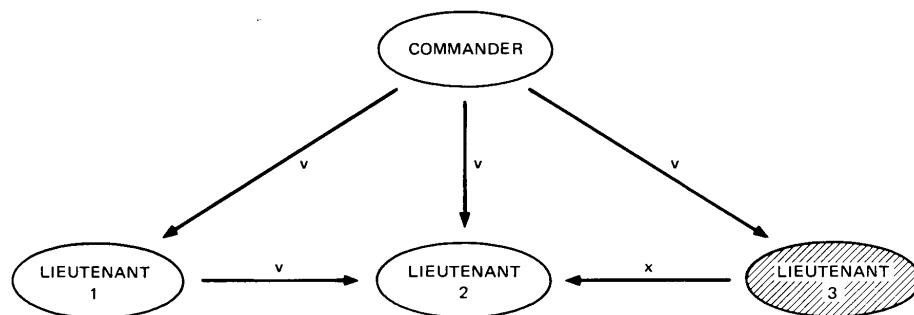


Figure 3 Algorithm OM(1); Lieutenant 3 a traitor.

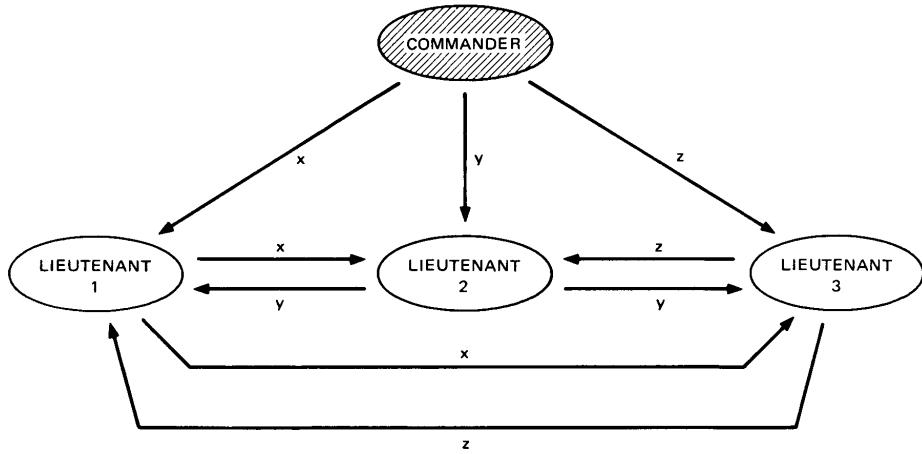


Figure 4 Algorithm OM(1); the commander a traitor.

values x , y , and z to the three lieutenants. Each lieutenant obtains $v_1 = x$, $v_2 = y$, and $v_3 = z$, so they all obtain the same value $\text{majority}(x, y, z)$ in step (3), regardless of whether or not any of the three values x , y , and z are equal.

The recursive algorithm OM(m) invokes $n - 1$ separate executions of the algorithm OM($m - 1$), each of which invokes $n - 2$ executions of OM($m - 2$), etc. This means that, for $m > 1$, a lieutenant sends many separate messages to each other lieutenant. There must be some way to distinguish among these different messages. The reader can verify that all ambiguity is removed if each lieutenant i prefixes the number i to the value v_i that he sends in step (2). As the recursion “unfolds,” the algorithm OM($m - k$) will be called $(n - 1) \cdots (n - k)$ times to send a value prefixed by a sequence of k lieutenants’ numbers.

To prove the correctness of the algorithm OM(m) for arbitrary m , we first prove the following lemma.

Lemma 1 For any m and k , Algorithm OM(m) satisfies IC2 if there are more than $2k + m$ generals and at most k traitors.

Proof The proof is by induction on m . IC2 only specifies what must happen if the commander is loyal. Using A1, it is easy to see that the trivial algorithm OM(0) works if the commander is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m - 1$, $m > 0$, and prove it for m .

In step (1), the loyal commander sends a value v to all $n - 1$ lieutenants. In step (2), each loyal lieutenant applies OM($m - 1$) with $n - 1$ generals. Since by hypothesis $n > 2k + m$, we have $n - 1 > 2k + (m - 1)$, so we can apply the induc-

tion hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j . Since there are at most k traitors, and $n - 1 > 2k + (m - 1) \geq 2k$, a majority of the $n - 1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n - 1$ values i , so he obtains $\text{majority}(v_1, \dots, v_{n-1}) = v$ in step (3), proving IC2. ■

The following theorem asserts that Algorithm OM(m) solves the Byzantine Generals Problem.

Theorem 1 For any m , Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more than $3m$ generals and at most m traitors.

Proof The proof is by induction on m . If there are no traitors, then it is easy to see that OM(0) satisfies IC1 and IC2. We therefore assume that the theorem is true for OM($m - 1$) and prove it for OM(m), $m > 0$.

We first consider the case in which the commander is loyal. By taking k equal to m in Lemma 1, we see that OM(m) satisfies IC2. IC1 follows from IC2 if the commander is loyal, so we need only verify IC1 in the case that the commander is a traitor.

There are at most m traitors, and the commander is one of them, so at most $m - 1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m - 1$ lieutenants, and $3m - 1 > 3(m - 1)$. We may therefore apply the induction hypothesis to conclude that OM($m - 1$) satisfies conditions IC1 and IC2. Hence, for each j , any two loyal lieutenants get the same value for v_j in step (3). (This follows from IC2 if one of the two lieutenants is Lieutenant j , and from IC1 otherwise.) Hence, any two loyal lieutenants get the same vector of values v_1, \dots, v_{n-1} , and therefore obtain the same value $\text{majority}(v_1, \dots, v_{n-1})$ in step (3), proving IC1. ■

4

A Solution with Signed Messages

As we saw from the scenario of Figures 1 and 2, it is the traitors' ability to lie that makes the Byzantine Generals Problem so difficult. The problem becomes easier to solve if we can restrict that ability. One way to do this is to allow the generals to send unforgeable signed messages. More precisely, we add to A1–A3 the following assumption:

- A4**
- (a) A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
 - (b) Anyone can verify the authenticity of a general's signature.

Note that we make no assumptions about a traitorous general's signature. In particular, we allow his signature to be forged by another traitor, thereby permitting collusion among the traitors.

Now that we have introduced signed messages, our previous argument that four generals are required to cope with one traitor no longer holds. In fact, a three-general solution does exist. We now give an algorithm that copes with m traitors for any number of generals. (The problem is vacuous if there are fewer than $m + 2$ generals.)

In our algorithm, the commander sends a signed order to each of his lieutenants. Each lieutenant then adds his signature to that order and sends it to the other lieutenants, who add their signatures and send it to others, and so on. This means that a lieutenant must effectively receive one signed message, make several copies of it, and sign and send those copies. It does not matter how these copies are obtained; a single message might be photocopied, or else each message might consist of a stack of identical messages which are signed and distributed as required.

Our algorithm assumes a function choice which is applied to a set of orders to obtain a single one. The only requirements we make for this function are

1. If the set V consists of the single element v , then $\text{choice}(V) = v$.
2. $\text{choice}(\emptyset) = \text{RETREAT}$, where \emptyset is the empty set.

Note that one possible definition is to let $\text{choice}(V)$ be the median element of V —assuming that there is an ordering of the elements.

In the following algorithm, we let $x : i$ denote the value x signed by General i . Thus, $v : j : i$ denotes the value v signed by j , and then that value $v : j$ signed by i . We let General 0 be the commander. In this algorithm, each lieutenant i maintains a set V_i , containing the set of properly signed orders he has received so far. (If the commander is loyal, then this set should never contain more than a single element.) Do not confuse V_i , the set of orders he has received, with the set of messages that he has received. There may be many different messages with the same order.

Algorithm SM(m) Initially $V_i = \emptyset$.

1. The commander signs and sends his value to every lieutenant.
2. For each i :
 - (A) If Lieutenant i receives a message of the form $v : 0$ from the commander and he has not yet received any order, then
 - (i) he lets V_i equal $\{v\}$;
 - (ii) he sends the message $v : 0 : i$ to every other lieutenant.

- (B) If Lieutenant i receives a message of the form $v : 0 : j_i : \dots : j_k$ and v is not in the set V_i , then
- he adds v to V_i ;
 - if $k < m$, then he sends the message $v : 0 : j_i : \dots : j_k : i$ to every lieutenant other than j_1, \dots, j_k .
3. For each i : When Lieutenant i will receive no more messages, he obeys the order $\text{choice}(V_i)$.

Note that in step (2), Lieutenant i ignores any message containing an order v that is already in the set V_i .

We have not specified how a lieutenant determines in step (3) that he will receive no more messages. By induction on k , one easily shows that for each sequence of lieutenants j_1, \dots, j_k with $k \leq m$, a lieutenant can receive at most one message of the form $v : 0 : j_1 : \dots : j_k$ in step (2). If we require that Lieutenant j_k either send such a message or else send a message reporting that he will not send such a message, then it is easy to decide when all messages have been received. (By assumption A3, a lieutenant can determine if a traitorous lieutenant j_k sends neither of those two messages.) Alternatively, time-out can be used to determine when no more messages will arrive. The use of time-out is discussed in Section 6.

Note that in step (2), Lieutenant i ignores any messages that do not have the proper form of a value followed by a string of signatures. If packets of identical messages are used to avoid having to copy messages, this means that he throws away any packet that does not consist of a sufficient number of identical, properly signed messages. (There should be $(n - k - 2)(n - k - 3) \dots (n - m - 2)$ copies of the message if it has been signed by k lieutenants.)

Figure 5 illustrates Algorithm SM(1) for the case of three generals when the commander is a traitor. The commander sends an “attack” order to one lieutenant and a “retreat” order to the other. Both lieutenants receive the two orders in step (2), so after step (2) $V_1 = V_2 = \{\text{“attack”}, \text{“retreat”}\}$, and they both obey the order $\text{choice}(\{\text{“attack”}, \text{“retreat”}\})$. Observe that here, unlike the situation in Figure 2, the lieutenants know the commander is a traitor because his signature appears on two different orders, and A4 states that only he could have generated those signatures.

In Algorithm SM(m), a lieutenant signs his name to acknowledge his receipt of an order. If he is the m th lieutenant to add his signature to the order, then that signature is not relayed to anyone else by its recipient, so it is superfluous. (More precisely, assumption A2 makes it unnecessary.) In particular, the lieutenants need not sign their messages in SM(1).

We now prove the correctness of our algorithm.

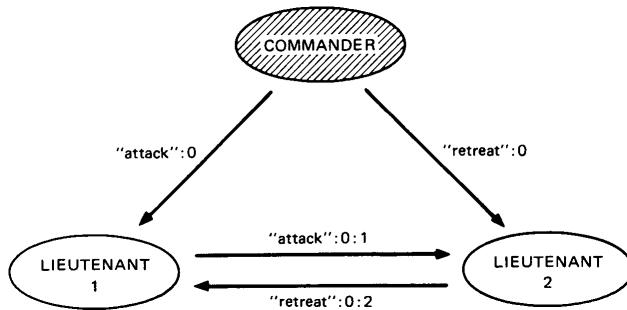


Figure 5 Algorithm SM(1); the commander a traitor.

Theorem 2 For any m , Algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors.

Proof We first prove IC2. If the commander is loyal, then he sends his signed order $v : 0$ to every lieutenant in step (1). Every loyal lieutenant will therefore receive the order v in step (2)(A). Moreover, since no traitorous lieutenant can forge any other message of the form $v' : 0$, a loyal lieutenant can receive no additional order in step (2)(B). Hence, for each loyal Lieutenant i , the set V_i obtained in step (2) consists of the single order v , which he will obey in step (3) by property 1 of the *choice* function. This proves IC2.

Since IC1 follows from IC2 if the commander is loyal, to prove IC1 we need only consider the case in which the commander is a traitor. Two loyal lieutenants i and j obey the same order in step (3) if the sets of orders V_i and V_j that they receive in step (2) are the same. Therefore, to prove IC1 it suffices to prove that, if i puts an order v into V_i in step (2), then j must put the same order v into V_j in step (2). To do this, we must show that j receives a properly signed message containing that order. If i receives the order v in step (2)(A), then he sends it to j in step (2)(A)(ii); so j receives it (by A1). If i adds the order to V_i in step (2)(B), then he must receive a first message of the form $v : 0 : j_1 : \dots : j_k$. If j is one of the j_r , then by A4 he must already have received the order v . If not, we consider two cases:

1. $k < m$. In this case, i sends the message $v : 0 : j_1 : \dots : j_k : i$ to j ; so j must receive the order v .
2. $k = m$. Since the commander is a traitor, at most $m - 1$ of the lieutenants are traitors. Hence, at least one of the lieutenants j_1, \dots, j_m is loyal. This loyal lieutenant must have sent j the value v when he first received it, so j must therefore receive that value.

This completes the proof. ■

5 Missing Communication Paths

Thus far, we have assumed that a general can send messages directly to every other general. We now remove this assumption. Instead, we suppose that physical barriers place some restrictions on who can send messages to whom. We consider the generals to form the nodes of a simple,² finite undirected graph G , where an arc between two nodes indicates that those two generals can send messages directly to one another. We now extend Algorithms OM(m) and SM(m), which assumed G to be completely connected, to more general graphs.

To extend our oral message algorithm OM(m), we need the following definition, where two generals are said to be *neighbors* if they are joined by an arc.

- Definition 1**
- (a) A set of nodes $\{i_1, \dots, i_p\}$ is said to be a *regular set of neighbors* of a node i if
 - (i) each i_j is a neighbor of i , and
 - (ii) for any general k different from i , there exist paths $\gamma_{j,k}$ from i_j to k not passing through i such that any two different paths $\gamma_{j,k}$ have no node in common other than k .
 - (b) The graph G is said to be *p-regular* if every node has a regular set of neighbors consisting of p distinct nodes.

Figure 6 shows an example of a simple 3-regular graph. Figure 7 shows an example of a graph that is not 3-regular because the central node has no regular set of neighbors containing three nodes.

We extend OM(m) to an algorithm that solves the Byzantine Generals Problem in the presence of m traitors if the graph G of generals is $3m$ -regular. (Note that a $3m$ -regular graph must contain at least $3m + 1$ nodes.) For all *positive* integers m and p , we define the algorithm OM(m, p) as follows when the graph G of generals

2. A simple graph is one in which there is at most one arc joining any two nodes, and every arc connects two distinct nodes.

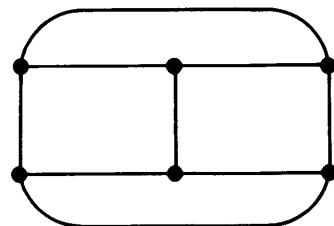


Figure 6 A 3-regular graph.

218 The Byzantine Generals Problem

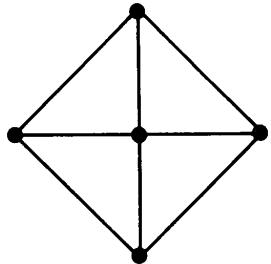


Figure 7 A graph that is not 3-regular.

is p -regular. ($\text{OM}(m, p)$ is not defined if G is not p -regular.) The definition uses induction on m .

Alg. $\text{OM}(m, p)$

0. Choose a regular set N of neighbors of the commander consisting of p lieutenants.
1. The commander sends his value to every lieutenant in N .
2. For each i in N , let v_i be the value Lieutenant i receives from the commander, or else RETREAT if he receives no value. Lieutenant i sends v_i to every other lieutenant k as follows:
 - (A) If $m = 1$, then by sending the value along the path $\gamma_{i,k}$ whose existence is guaranteed by part (a)(ii) of Definition 1.
 - (B) If $m > 1$, then by acting as the commander in the algorithm $\text{OM}(m - 1, p - 1)$, with the graph of generals obtained by removing the original commander from G .
3. For each k , and each i in N with $i \neq k$, let v_i be the value Lieutenant k received from Lieutenant i in step (2), or RETREAT if he received no value. Lieutenant k uses the value $\text{majority}(v_{i_1}, \dots, v_{i_p})$, where $N = \{i_1, \dots, i_p\}$.

Note that removing a single node from a p -regular graph leaves a $(p - 1)$ -regular graph. Hence, one can apply the algorithm $\text{OM}(m - 1, p - 1)$ in step (2)(B).

We now prove that $\text{OM}(m, 3m)$ solves the Byzantine Generals Problem if there are at most m traitors. The proof is similar to the proof for the algorithm $\text{OM}(m)$ and will just be sketched. It begins with the following extension of Lemma 1.

Lemma 2 For any $m > 0$ and any $p \geq 2k + m$, Algorithm $\text{OM}(m, p)$ satisfies IC2 if there are at most k traitors.

Proof For $m = 1$, observe that a lieutenant obtains the value $\text{majority}(v_1, \dots, v_p)$, where each v_i is a value sent to him by the commander along a path disjoint from the path used to send the other values to him. Since there are at most k traitors and $p \geq 2k + 1$, more than half of those paths are composed entirely of loyal lieutenants. Hence, if the commander is loyal, then a majority of the values v_i will equal the value he sent, which implies that IC2 is satisfied.

Now assume the lemma for $m - 1$, $m > 1$. If the commander is loyal, then each of the p lieutenants in N gets the correct value. Since $p > 2k$, a majority of them are loyal, and by the induction hypothesis each of them sends the correct value to every loyal lieutenant. Hence, each loyal lieutenant gets a majority of correct values, thereby obtaining the correct value in step (3). ■

The correctness of Algorithm OM($m, 3m$) is an immediate consequence of the following result.

Theorem 3 For any $m > 0$ and any $p \geq 3m$, Algorithm OM(m, p) solves the Byzantine Generals Problem if there are at most m traitors.

Proof By Lemma 2, letting $k = m$, we see that OM(m, p) satisfies IC2. If the commander is loyal, then IC1 follows from IC2, so we need only prove IC1 under the assumption that the commander is a traitor. To do this, we prove that every loyal lieutenant gets the same set of values v_i in step (3). If $m = 1$, then this follows because all the lieutenants, including those in N , are loyal and the paths $\gamma_{i,k}$ do not pass through the commander. For $m > 1$, a simple induction argument can be applied, since $p \geq 3m$ implies that $p - 1 \geq 3(m - 1)$. ■

Our extension of Algorithm OM(m) requires that the graph G be $3m$ -regular, which is a rather strong connectivity hypothesis.³ In fact, if there are only $3m + 1$ generals (the minimum number required), then $3m$ -regularity means complete connectivity, and Algorithm OM($m, 3m$) reduces to Algorithm OM(m). In contrast, Algorithm SM(m) is easily extended to allow the weakest possible connectivity hypothesis. Let us first consider how much connectivity is needed for the Byzantine Generals Problem to be solvable. IC2 requires that a loyal lieutenant obey a loyal commander. This is clearly impossible if the commander cannot communicate with the lieutenant. In particular, if every message from the commander to the lieutenant must be relayed by traitors, then there is no way to guarantee that the lieutenant gets the commander's order. Similarly, IC1 cannot be guaranteed if there

3. A recent algorithm of Dolev [2] requires less connectivity.

are two lieutenants who can only communicate with one another via traitorous intermediaries.

The weakest connectivity hypothesis for which the Byzantine Generals Problem is solvable is that the subgraph formed by the loyal generals be connected. We show that under this hypothesis, the algorithm SM($n - 2$) is a solution, where n is the number of generals—regardless of the number of traitors. Of course, we must modify the algorithm so that generals only send messages to where they can be sent. More precisely, in step (1), the commander sends his signed order only to his neighboring lieutenants; and in step (2)(B), Lieutenant i only sends the message to every *neighboring* lieutenant not among the j_r .

We prove the following more general result, where the diameter of a graph is the smallest number d such that any two nodes are connected by a path containing at most d arcs.

Theorem 4 For any m and d , if there are at most m traitors and the subgraph of loyal generals has diameter d , then Algorithm SM($m + d - 1$) (with the above modification) solves the Byzantine Generals Problem.

Proof The proof is quite similar to that of Theorem 2 and is just sketched here. To prove IC2, observe that by hypothesis there is a path from the loyal commander to a lieutenant i going through $d - 1$ or fewer loyal lieutenants. Those lieutenants will correctly relay the order until it reaches i . As before, assumption A4 prevents a traitor from forging a different order.

To prove IC1, we assume the commander is a traitor and must show that any order received by a loyal lieutenant i is also received by a loyal lieutenant j . Suppose i receives an order $v : 0 : j_1 : \dots : j_k$ not signed by j . If $k < m$, then i will send it to every neighbor who has not already received that order, and it will be relayed to j within $d - 1$ more steps. If $k \geq m$, then one of the first m signers must be loyal and must have sent it to all of his neighbors, whereupon it will be relayed by loyal generals and will reach j within $d - 1$ steps. ■

Corollary If the graph of loyal generals is connected, then SM($n - 2$) (as modified above) solves the Byzantine Generals Problem for n generals.

Proof Let d be the diameter of the graph of loyal generals. Since the diameter of a connected graph is less than the number of nodes, there must be more than d loyal generals and fewer than $n - d$ traitors. The result follows from the theorem by letting $m = n - d - 1$. ■

Theorem 4 assumes that the subgraph of loyal generals is connected. Its proof is easily extended to show that even if this is not the case, if there are at most m traitors, then the algorithm SM($m + d - 1$) has the following two properties:

1. Any two loyal generals connected by a path of length at most d passing through only loyal generals will obey the same order.
2. If the commander is loyal, then any loyal lieutenant connected to him by a path of length at most $m + d$ passing only through loyal generals will obey his order.

6 Reliable Systems

Other than using intrinsically reliable circuit components, the only way we know to implement a reliable computer system is to use several different “processors” to compute the same result, and then to perform a majority vote on their outputs to obtain a single value. (The voting may be performed within the system, or externally by the users of the output.) This is true whether one is implementing a reliable computer using redundant circuitry to protect against the failure of individual chips, or a ballistic missile defense system using redundant computing sites to protect against the destruction of individual sites by a nuclear attack. The only difference is in the size of the replicated “processor”.

The use of majority voting to achieve reliability is based upon the assumption that all the nonfaulty processors will produce the same output. This is true so long as they all use the same input. However, any single input datum comes from a single physical component—for example, from some other circuit in the reliable computer, or from some radar site in the missile defense system—and a malfunctioning component can give different values to different processors. Moreover, different processors can get different values even from a nonfaulty input unit if they read the value while it is changing. For example, if two processors read a clock while it is advancing, then one may get the old time and the other the new time. This can only be prevented by synchronizing the reads with the advancing of the clock.

In order for majority voting to yield a reliable system, the following two conditions should be satisfied:

1. All nonfaulty processors must use the same input value (so they produce the same output).
2. If the input unit is nonfaulty, then all nonfaulty processes use the value it provides as input (so they produce the correct output).

These are just our interactive consistency conditions IC1 and IC2, where the “commander” is the unit generating the input, the “lieutenants” are the processors, and “loyal” means nonfaulty.

It is tempting to try to circumvent the problem with a “hardware” solution. For example, one might try to insure that all processors obtain the same input value by having them all read it from the same wire. However, a faulty input unit could send a marginal signal along the wire—a signal that can be interpreted by some processors as a 0 and by others as a 1. There is no way to guarantee that different processors will get the same value from a possibly faulty input device except by having the processors communicate among themselves to solve the Byzantine Generals Problem.

Of course, a faulty input device may provide meaningless input values. All that a Byzantine Generals solution can do is guarantee that all processors use the same input value. If the input is an important one, then there should be several separate input devices providing redundant values. For example, there should be redundant radars as well as redundant processing sites in a missile defense system. However, redundant inputs cannot achieve reliability; it is still necessary to insure that the nonfaulty processors use the redundant data to produce the same output.

In case the input device is nonfaulty but gives different values because it is read while its value is changing, we still want the nonfaulty processors to obtain a reasonable input value. It can be shown that, if the functions *majority* and *choice* are taken to be the median functions, then our algorithms have the property that the value obtained by the nonfaulty processors lies within the range of values provided by the input unit. Thus, the nonfaulty processors will obtain a reasonable value so long as the input unit produces a reasonable range of values.

We have given several solutions, but they have been stated in terms of Byzantine generals rather than in terms of computing systems. We now examine how these solutions can be applied to reliable computing systems. Of course, there is no problem implementing a “general’s” algorithm with a processor. The problems lie in implementing a message passing system that meets assumptions A1–A3 (assumptions A1–A4 for Algorithm SM(m)). We now consider these assumptions in order.

A1. Assumption A1 states that every message sent by a nonfaulty processor is delivered correctly. In real systems, communication lines can fail. For the oral message algorithms OM(m) and OM(m, p), the failure of the communication line joining two processors is indistinguishable from the failure of one of the processors. Hence, we can only guarantee that these algorithms will work in the presence of up to m failures, be they processor or communication line failures. (Of course,

the failure of several communication lines attached to the same processor is equivalent to a single processor failure.) If we assume that a failed communication line cannot result in the forgery of a signed message—an assumption which we will see below is quite reasonable, then our signed message algorithm $SM(m)$ is insensitive to communication line failure. More precisely, Theorem 4 remains valid even with communication line failure. A failed communication line has the same effect as simply removing the communication line—it lowers the connectivity of the processor graph.

A2. Assumption A2 states that a processor can determine the originator of any message that it received. What is actually necessary is that a faulty processor not be able to impersonate a nonfaulty one. In practice, this means that interprocess communication be over fixed lines rather than through some message switching network. (If a switching network is used, then faulty network nodes must be considered, and the Byzantine Generals Problem appears again.) Note that assumption A2 is not needed if A4 is assumed and all messages are signed, since impersonation of another processor would imply forging its messages.

A3. Assumption A3 requires that the absence of a message can be detected. The absence of a message can only be detected by its failure to arrive within some fixed length of time—in other words, by the use of some time-out convention. The use of time-out to satisfy A3 requires two assumptions:

1. There is a fixed maximum time needed for the generation and transmission of a message.
2. The sender and receiver have clocks that are synchronized to within some fixed maximum error.

The need for the first assumption is fairly obvious, since the receiver must know how long he needs to wait for the message to arrive. (The generation time is how long it takes the processor to send the message after receiving all the input necessary to generate it.) The need for the second assumption is less obvious. However, it can be shown that either this assumption or an equivalent one is necessary to solve the Byzantine Generals Problem. More precisely, suppose that we allow algorithms in which the generals take action only in the following circumstances:

1. At some fixed initial time (the same for all generals).
2. Upon the receipt of a message.
3. When a randomly chosen length of time has elapsed. (I.e., a general can set a timer to a random value and act when the timer goes off.)

(This yields the most general class of algorithms we can envision which does not allow the construction of synchronized clocks.) It can be shown that no such algorithm can solve the Byzantine Generals Problem if messages can be transmitted arbitrarily quickly, even if there is an upper bound on message transmission delay. Moreover, no solution is possible even if we restrict the traitors so that the only incorrect behavior they are permitted is the failure to send a message. The proof of this result is beyond the scope of this paper. Note that placing a lower as well as an upper bound on transmission delay allows processors to implement clocks by sending messages back and forth.

The above two assumptions make it easy to detect unsent messages. Let μ be the maximum message generation and transmission delay, and assume the nonfaulty processors have clocks that differ from one another by at most τ at any time. Then any message that a nonfaulty process should begin to generate by time T on its clock will arrive at its destination by time $T + \mu + \tau$ on the receiver's clock. Hence, if the receiver has not received the message by that time, then it may assume that it was not sent. (If it arrives later, then the sender must be faulty, so the correctness of our algorithms does not depend upon the message being sent.) By fixing the time at which the input processor sends its value, one can calculate until what time on its own clock a processor must wait for each message. For example, in Algorithm SM(m) a processor must wait until time $T_0 + k(\mu + \tau)$ for any message having k signatures, where T_0 is the time (on his clock) at which the commander starts executing the algorithm.

No two clocks run at precisely the same rate, so no matter how accurately the processors' clocks are synchronized initially, they will eventually drift arbitrarily far apart unless they are periodically resynchronized. We therefore have the problem of keeping the processors' clocks all synchronized to within some fixed amount, even if some of the processors are faulty. This is as difficult a problem as the Byzantine Generals Problem itself. Solutions to the clock synchronization problem exist which are closely related to our Byzantine Generals solutions. They will be described in a future paper.

A4. Assumption A4 requires that processors be able to sign their messages in such a way that a nonfaulty processor's signature cannot be forged. A signature is a piece of redundant information $S_i(M)$ generated by process i from a data item M . A message signed by i consists of a pair $(M, S_i(M))$. To meet parts (a) and (b) of A4, the function S_i must have the following two properties:

- (a) If processor i is nonfaulty, then no faulty processor can generate $S_i(M)$.
- (b) Given M and X , any process can determine if X equals $S_i(M)$.

Property (a) can never be guaranteed, since $S_i(M)$ is just a data item, and a faulty processor could generate any data item. However, we can make the probability of its violation as small as we wish, thereby making the system as reliable as we wish. How this is done depends upon the type of faults we expect to encounter. There are two cases of interest:

1. *Random Malfunction.* By making S_i a suitably “randomizing” function, we can make the probability that a random malfunction in a processor generates a correct signature essentially equal to the probability of its doing so through a random choice procedure—that is, the reciprocal of the number of possible signatures. The following is one method for doing this. Assume that messages are encoded as positive integers less than P , where P is a power of two. Let $S_i(M)$ equal $M * K_i \bmod P$, where K_i is a randomly chosen odd number less than P . Letting K_i^{-1} be the unique number less than P such that $K_i * K_i^{-1} \equiv 1 \bmod P$, a process can check that $X = S_i(M)$ by testing that $M \equiv X * K_i^{-1} \bmod P$. If another processor does not have K_i in its memory, then the probability of its generating the correct signature $M * K_i$ for a single (nonzero) message M should be $1/P$: its probability of doing so by random choice. (Note that if the processor could obtain K_i by some simple procedure, then there might be a larger probability of a faulty processor j forging i ’s signature by substituting K_i for K_j when trying to compute $S_j(M)$.)
2. *Malicious Intelligence.* If the faulty processor is being guided by a malicious intelligence—for example, if it is a perfectly good processor being operated by a human who is trying to disrupt the system—then the construction of the signature function S_i becomes a cryptography problem. We refer the reader to [1] and [4] for a discussion of how this problem can be solved.

Note that it is easy to generate the signature $S_i(M)$ if the process has already seen that signature. Hence, it is important that the same message never have to be signed twice. This means that, when using $SM(m)$ repeatedly to distribute a sequence of values, sequence numbers should be appended to the values to guarantee uniqueness.

7 Conclusion

We have presented several solutions to the Byzantine Generals Problem, under various hypotheses, and shown how they can be used in implementing reliable computer systems. These solutions are expensive in both the amount of time and the number of messages required. Algorithms $OM(m)$ and $SM(m)$ both require

message paths of length up to $m + 1$. In other words, each lieutenant may have to wait for messages that originated at the commander and were then relayed via m other lieutenants. Fischer and Lynch have shown that this must be true for any solution that can cope with m traitors, so our solutions are optimal in that respect. Our algorithms for a graph that is not completely connected require message paths of length up to $m + d$, where d is the diameter of the subgraph of loyal generals. We suspect that this is also optimal.

Algorithms OM(m) and SM(m) involve sending up to $(n - 1)(n - 2) \dots (n - m - 1)$ messages. The number of separate messages required can certainly be reduced by combining messages. It may also be possible to reduce the amount of information transferred, but this has not been studied in detail. However, we expect that a large number of messages will still be required.

Achieving reliability in the face of arbitrary malfunctioning is a difficult problem, and its solution seems to be inherently expensive. The only way to reduce the cost is to make assumptions about the type of failure that may occur. For example, it is often assumed that a computer may fail to respond but will never respond incorrectly. However, when extremely high reliability is required, such assumptions cannot be made, and the full expense of a Byzantine Generals solution is required.

References

- [1] DIFFIE, W., AND HELLMAN, M.E. New directions in cryptography. *IEEE Trans. Inf. Theory* IT-22 (Nov. 1976), 644–654.
- [2] DOLEV, D. The Byzantine generals strike again. *J. Algorithms* 3, 1 (Jan. 1982).
- [3] PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM* 27, 2 (Apr. 1980), 228–234.
- [4] RIVEST, R.L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126.

Received April 1980; revised November 1981; accepted November 1981

The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication

Leslie Lamport (Digital Equipment Corporation)

Abstract

A novel formal theory of concurrent systems that does not assume any atomic operations is introduced. The execution of a concurrent program is modeled as an abstract set of operation executions with two temporal ordering relations: “precedence” and “can causally affect”. A primitive interprocess communication mechanism is then defined. In Part II, the mutual exclusion is expressed precisely in terms of this model, and solutions using the communication mechanism are given.

Categories and Subject Descriptors: B.3.m [**Memory Structures**]: Miscellaneous; B.4.m [**Input/Output and Data Communications**]: Miscellaneous; D.4.1 [**Operating Systems**]: Process Management—*concurrency; mutual exclusion*; F.3.m [**Logics and Meanings of Programs**]: Miscellaneous

Most of this work was performed while the author was at SRI International, where it was supported in part by the National Science Foundation under grant number MCS 78-16783.

Author's address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1986 ACM 0004-5411/86/0400-0313 \$00.75

Paper originally published in *Journal of the Association for Computing Machinery*, 33(2), April 1986, pp. 313–326.

General Terms: Theory

Additional Key Words and Phrases: Nonatomic operations, readers/writers, shared variables

1 Introduction

The mutual exclusion problem was first described and solved by Dijkstra in [3]. In this problem, there is a collection of asynchronous processes, each alternately executing a critical and a noncritical section, that must be synchronized so that no two processes ever execute their critical sections concurrently. Mutual exclusion lies at the heart of most concurrent process synchronization and, apart from its practical motivation, the mutual exclusion problem is of great theoretical significance.

The concept of mutual exclusion is deeply ingrained in the way computer scientists think about concurrency. Almost all formal models of concurrent processing are based upon an underlying assumption of mutually exclusive atomic operations, and almost all interprocess communication mechanisms that have been proposed require some underlying mutual exclusion in their implementation. Hence, these models and mechanisms are not satisfactory for a fundamental study of the mutual exclusion problem. We have therefore been forced to develop a new formalism for talking about concurrent systems, and a new way of viewing interprocess communication. Part I is entirely devoted to this formalism, which we believe provides a basis for discussing other fundamental problems in concurrent processing as well; the mutual exclusion problem itself is discussed in Part II [9].

The formal model we have developed is radically different from commonly used ones, and will appear strange to computer scientists accustomed to thinking in terms of atomic operations. (It is a slight extension to the one we introduced in [6].) When diverging from the beaten path in this way, one is in great danger of becoming lost in a morass of irrelevance. To guard against this, we have continually used physical reality as our guidepost. (Perhaps this is why hardware designers seem to understand our ideas more easily than computer scientists.) We therefore give a very careful physical justification for all the definitions and axioms in our formalism. Although this is quite unusual in theoretical computer science, we feel that it is necessary in explaining and justifying our departure from the traditional approach.

2 The Model

We begin by describing a formal model in which to state the problem and the solution. Except for the one introduced by us in [6], all formal models of concurrent

processes that we know of are based upon the concept of an indivisible atomic operation. The concurrent execution of any two atomic operations is assumed to have the same effect as executing them in some order. However, if two operations can affect one another—e.g., if they perform interprocess communication—then implementing them to be atomic is equivalent to making the two operations mutually exclusive. Hence, assuming atomic operations is tantamount to assuming a lower-level solution to the mutual exclusion problem. Any algorithm based upon atomic operations cannot be considered a fundamental solution to the mutual exclusion problem. We therefore need a formalism that is not based upon atomic operations. The one we use is a slight extension to the formalism of [6].

2.1 Physical Considerations

For our results to be meaningful, our formalism must accurately reflect the physical reality of concurrent processes. We therefore feel that it is important to justify the formalism on physical grounds. We do this in terms of the geometry of space-time, which lies at the foundation of all modern physics. We begin with a brief exposition of this geometry. A more thorough exposition can be found in [15] and [16], but for the more sophisticated reader we recommend the original works [4,11].

The reader may find the introduction of special relativity a bit far-fetched, since one is rarely, if ever, concerned with systems of processes moving at relativistic velocities relative to one another. However, the relativistic view of time is relevant whenever signal propagation time is not negligibly small compared to the execution time of individual operations, and this is certainly the case in most multiprocess systems.

Because it is difficult to draw pictures of four-dimensional space-time, we will discuss a three-dimensional space-time for a two-dimensional spatial universe. Everything generalizes easily to four-dimensional space-time.¹ We picture space-time as a three-dimensional Cartesian space whose points are called *events*, where the point (x, y, t) is the event occurring at time t at the point with spatial coordinates (x, y) . Dimensional units are chosen so the speed of light equals 1.

The *world line* of a point object is the locus of all events (x, y, t) such that the object is at location (x, y) at time t . Since photons travel in a straight line with speed 1, the world line of a photon is a straight line inclined at 45° to the x - y plane. The forward light cone emanating from an event e is the surface formed by all possible world lines of photons created at that event. This is illustrated in

1. While it is even easier to draw pictures of a two-dimensional space-time with a single space dimension, a one-dimensional space has some special properties (such as the ability to send a light beam in only two directions) that can make such pictures misleading.

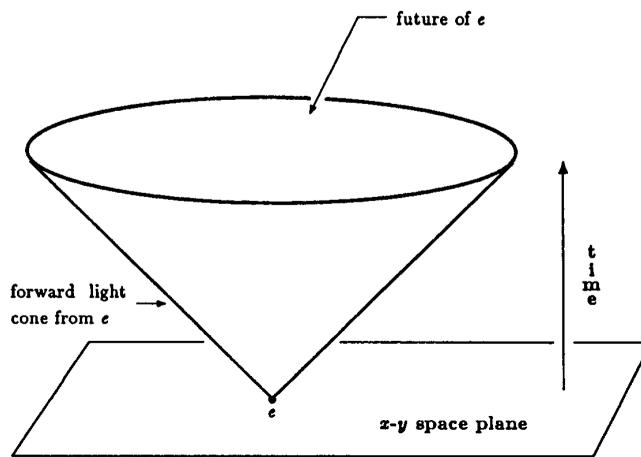


Figure 1 Space-time.

Figure 1. The *future* of event e consists of all events other than e itself that lie on or inside the future light cone emanating from e . It is a fundamental principle of special relativity that an event e can only influence the events in its future.

We say that an event e *precedes* an event f , written $e \rightarrow f$, if f lies in the future of e . It is easy to see that \rightarrow is an irreflexive partial ordering—i.e., that (i) $e \not\rightarrow e$ and (ii) $e \rightarrow f \rightarrow g$ implies $e \rightarrow g$. Two events are said to be *concurrent* if neither precedes the other. Since objects cannot travel faster than light, two different events lying on the world-line of an object cannot be concurrent.

We can think of the vertical line through the origin as the world line of some standard clock, where the event $(0, 0, t)$ on this world line represents the clock “striking” time t . A horizontal plane, consisting of all events having the same t -coordinate, represents the universe at time t —as viewed by us. However, another observer may have a different view of which events occur at time t . We define a *space-like* plane to be a plane making an angle of less than 45° with the x - y plane. For an inertial observer, the set of events occurring at time t forms a space-like plane through $(0, 0, t)$. (An inertial observer is one traveling in a straight line at constant speed.) For different values of t , these planes are parallel (for the same observer). Any space-like plane represents a set of events that some inertial observer regards as all occurring at the same time. Such a plane defines an obvious partitioning of space-time into three sets: the future, the past, and the present (the plane itself).

It follows from these observations that an event e precedes an event f if and only if every inertial observer regards e as happening before f , and events e and f

are concurrent if and only if there is some observer who views them as happening at the same time.

2.2 System Executions

According to classical physics, the universe consists of a continuum of space-time events, each having no spatial or temporal extent. In computer science, one imposes a discrete structure upon this continuous universe, considering a system to consist of distinct *operation executions* such as reading a flip-flop or sending a message.² An infinite (usually bounded) set of space-time events is considered to be a single operation execution. For example, the operation execution of reading a flip-flop consists of events spatially located at the flip-flop and perhaps at some of the wires connected to it.

The boundary between the events of one operation execution and of other operation executions in the same processor is rather arbitrary; events occurring along the wire leading from the flip-flop can be included as part of the reading of the flip-flop or as part of a subsequent operation execution that uses the value that was read. The fine details of where the boundary is drawn do not matter; extending the region of space-time comprising the operation execution by a nanosecond here or a micron there makes no difference. However, the choice of which events belong to which operation executions can influence the properties we ascribe to the operations; the formalism used to describe a system can depend upon whether the events in the propagation of a value along a wire belong to the send or to the receive operation.

An execution of a system therefore consists of a set of operation executions, where each operation execution consists of a nonempty set of space-time events. We define the relations \longrightarrow and \dashrightarrow on the set of operation executions as follows:

$$\begin{aligned} A \longrightarrow B &\stackrel{\text{def}}{=} \forall a \in A : \forall b \in B : a \longrightarrow b, \\ A \dashrightarrow B &\stackrel{\text{def}}{=} \exists a \in A : \exists b \in B : a \longrightarrow b \quad \text{or} \quad a = b. \end{aligned}$$

Thus, $A \longrightarrow B$ means that every event of A precedes every event of B , and $A \dashrightarrow B$ means that some event of A either precedes or is the same as some event of B . (If a read of a flip-flop occurs while the flip-flop is also being set, some space-time events located at the flip-flop may belong to both operation executions.)

2. Since the term “operation” often denotes a type of action that can be performed repeatedly, as in “the operation of addition”, we write “operation execution” to emphasize that we are referring to a single instance of such an action.

Remembering the meaning of the precedence relation for events, we read $A \rightarrow B$ as “ A precedes B ”, and $A \rightarrowtail B$ as “ A can causally affect B ”. However, we think of “can causally affect” as a purely temporal relation, independent of what the operations are doing. Thus, $A \rightarrowtail B$ can hold even if A and B are *read* operations that cannot actually influence one another. We say that A and B are *concurrent* if $A \not\rightarrow B$ and $B \not\rightarrow A$. In other words, two operation executions are concurrent unless one precedes the other.

The following properties of the relations \rightarrow and \rightarrowtail on operation executions follow directly from the fact that the relation \rightarrow on events is an irreflexive partial ordering:

- A1. The relation \rightarrow is an irreflexive partial ordering.
- A2. If $A \rightarrow B$, then $A \rightarrowtail B$ and $B \not\rightarrow A$.
- A3. If $A \rightarrow B \rightarrowtail C$ or $A \rightarrowtail B \rightarrow C$, then $A \rightarrowtail C$.
- A4. If $A \rightarrow B \rightarrowtail C \rightarrow D$, then $A \rightarrow D$.

There are two kinds of operation executions—terminating ones, whose events all occur before some time (they are in the past of some space-like surface), and nonterminating ones that go on forever (their events do not lie in the past of any space-like surface). We make the following assumptions about these two classes of operation executions.

- A5. For any terminating A , the set of B such that $A \not\rightarrow B$ is finite.
- A6. For any nonterminating A :
 - (a) The set of B such that $B \rightarrow A$ is finite.
 - (b) For all B : $A \not\rightarrow B$.

Properties A5 and A6 can be derived from the following assumptions.

- At any time, there are only a finite number of operation executions that have begun by that time—i.e., for any space-like surface, there are only a finite number of operation executions containing events in the past of that surface.
- There are only a finite number of operation executions concurrent with any terminating operation execution.

The second assumption means that the speed with which the system is “spreading out” in space is bounded by some value less than the speed of light.

We have described operation executions in terms of events in order to justify A1–A6. In computer science, one ignores the space-time events that comprise operation

executions. A programmer does not care that machine instructions are composed of more primitive events. In our formalism, operation executions are considered primitives elements, and A1–A6 are taken as axioms. We define a *system execution* to consist of a set of operation executions, partitioned into terminating and non-terminating ones, together with relations \rightarrow and $\rightarrow\rightarrow$ that satisfy Axioms A1–A6.

2.3 Higher-Level Views

A system can be viewed at many different levels; the programmer may consider the execution of a *load accumulator from memory* instruction to be a single operation, while the hardware designer considers it to be a sequence of lower-level register-transfer operations. The fundamental task in computing is to implement higher-level operations with lower-level ones. A hardware designer implements machine-language operations with register-transfer operations; a compiler writer implements Pascal operations with machine-language operations; and an applications programmer implements funds-transferring operations with Pascal operations. One assumes that the lower-level, primitive operations are given and uses them to construct the higher-level ones.

A higher-level operation execution consists of a set of lower-level ones. If we view operation executions as sets of space-time events, a higher-level operation execution is the union of the events of the (lower-level) operation executions it is composed of. It is nonterminating if and only if it consists of a finite number of nonterminating operation executions. It is not hard to show that the relations \rightarrow and $\rightarrow\rightarrow$ between higher-level operation executions can be computed from those relations between the lower-level operation executions as follows:

$$\begin{aligned} R \rightarrow S &= \forall A \in R : \forall B \in S : A \rightarrow B, \\ R \rightarrow\rightarrow S &= \exists A \in R : \exists B \in S : A \rightarrow\rightarrow B \quad \text{or} \quad A = B. \end{aligned} \tag{2.1}$$

Since events do not appear in our formalism, we cannot proceed in this way. Instead, we take (2.1) to be the definition of the relations \rightarrow and $\rightarrow\rightarrow$ between any two sets of operation executions. By identifying an operation execution A with the set $\{A\}$, this definition also applies when R or S is a single operation execution rather than a set of them. A set of operation executions is defined to be *terminating* if and only if it consists of a finite number of terminating operation executions.

Given a system execution, a higher-level view of that execution consists of a partitioning of its operation executions into sets, which represent higher-level operation executions. The machine-language view of a system is obtained by partitioning the register-transfer operations into executions of machine-language instructions. This need not be a true partition; a single register-transfer operation could be part

of the execution of two separate machine-language instructions. We therefore define a *higher-level view* of a system execution to be a collection \mathcal{H} of nonempty sets of operation executions such that each operation execution belongs to a finite number, greater than zero, of sets in \mathcal{H} . The elements of \mathcal{H} (which are sets of operation executions) are called the operation executions of the higher-level view, or simply the *higher-level operation executions*.

Given a higher-level view of a system execution, we have defined the relations \rightarrow and \dashrightarrow (by (2.1)) and the concept of termination on its high-level operation executions. Using these definitions and Axioms A1–A6 for the (lower-level) operation executions, it is easy to show that A1–A6 hold for the higher-level operation executions. Hence, the higher-level view of a system execution is itself a system execution.

In any study of computer systems, there is a lowest-level view that is of interest. The operation executions in that view will be called *elementary* operation executions. A set of elementary operation executions will be called an operation execution. (It is an operation execution in some higher-level view.)

3 Interprocess Communication

To achieve mutual exclusion, processes must be able to communicate with one another. We must therefore assume some interprocess communication mechanism. However, almost every communication primitive that has been proposed implicitly assumes mutual exclusion. For example, the first mutual exclusion algorithms assumed a central memory that can be accessed by all the processes, in which any two operations to a single memory cell occur in some definite order. In other words, they assumed mutually exclusive access to a memory cell. We will define an interprocess communication mechanism that does not assume any lower-level mutual exclusion. In order to explain our choice of a mechanism, we begin by examining the nature of interprocess communication.

The simplest form of interprocess communication is for a process i to send one bit of information to a process j . This can be done in two ways: by sending a message or by setting a bit. For example, if the physical communication medium is a wire, then “sending a message” might mean sending a pulse and “setting a bit” might mean setting a level. However, a message is a transient phenomenon, and j must be waiting for i ’s message in order to be sure of receiving it. We now show that with only this kind of transient communication, the mutual exclusion problem does not admit a solution in which the following two conditions hold:

- A process need communicate only when trying to enter or leave its critical section, not in its critical or noncritical sections.
- A process may remain forever in its noncritical section.

These conditions rule out algorithms in which processes take turns entering, or declining to enter, their critical section; such algorithms are really solutions to the producer/consumer problem [1].

Assume that a process i wants to enter its critical section first, while another process j is in its noncritical section. Since j could remain in its noncritical section forever, i must be able to enter its critical section without communicating with j . Assume that this has happened and i is in its critical section when j decides it wants to enter its critical section. Since i is not required to communicate while in its critical section, j cannot find out if i is in its critical section until i leaves the critical section. However, j cannot wait for a communication because i might be in, and remain forever in, its noncritical section. Hence, no solution is possible.

This conclusion is based upon the assumption that communication by transient messages can only be achieved if the receiving process is waiting for the message. This assumption may seem paradoxical since distributed systems often provide a message-passing facility with which a process can receive messages while engaged in other activity. A closer examination of such systems reveals that the receiving process actually consists of two concurrently executing subprocesses: a main subprocess that performs the process's major activity, and a communication subprocess that receives messages and stores them in a buffer to be read by the main subprocess, where one or more bits in the buffer may signal the main subprocess that it should interrupt its activity to process a message. The activity of the communication subprocess can be regarded as part of the sending operation, which effects the communication by setting bits in the buffer that can be read by the receiving process. Thus, this kind of message passing really involves the setting of bits at the remote site by the sender.

Hence, we assume that a process i communicates one bit of information to a process j by setting a communication bit that j can read. A bit that can be set but not reset can be used only once. Since there is no bound on the number of times a process may need to communicate, interprocess synchronization is impossible with a finite number of such "once only" communication bits. Therefore, we require that it be possible to reset the bit. This gives us three possibilities:

1. Only the reader can reset the communication bit.

2. Only the writer can reset the communication bit.
3. Both can reset the communication bit.

In case 1, with a finite number of bits, a process i can send only a bounded amount of information to another process j before j resets the bits. However, in the mutual exclusion problem, a process may spend arbitrarily long in its noncritical section, so process i can enter its critical section arbitrarily many times while process j is in its noncritical section. An argument similar to the one demonstrating that transient communication cannot be used shows that process i must communicate with process j every time it executes its critical section, so i may have to send an unbounded amount of information to j while j is in its noncritical section. Since a process need not communicate while in its noncritical section, the problem cannot be solved using the first kind of communication bit.³

Of the remaining two possibilities, we choose number 2 because it is more primitive than number 3. We are therefore led to the use of a communication bit that can be set to either of two values by one process and read by another—i.e., a boolean-valued communication variable with one writer and one reader. We let *true* and *false* denote the two values. We say that such a variable “belongs to” the process that can write it.

We now define the semantics of the operations of reading and writing a communication variable. A write operation execution for a communication variable has the form *write* $v := v'$, where v is the name of the variable, and v' denotes the value being written—either *true* or *false*. A read operation execution has the form *read* $v = v'$, where v' is the value obtained as the result of performing the read.

The first assumptions we make are:

- C0. Reads and writes are terminating operation executions.
- C1. A read of a communication variable obtains either the value *true* or the value *false*.

Physically, C1 means that no matter what state the variable is in when it is being read, the reader will interpret that state as one of the two possible values.

We require that all writes to a single communication variable be totally ordered by the \rightarrow relation. Since all of these writes are executed by the same process—the one that owns the variable—this is a reasonable requirement. We will see below that

3. However, it is possible to solve producer/consumer problems with it. In fact, an interrupt bit of an ordinary computer is precisely this kind of communication bit, and it is used to implement producer/consumer synchronization with its peripheral devices.

it is automatically enforced by the programming language in which the algorithms are described. This requirement allows us to introduce the following notation.

Definition 1 For any variable v we let $V^{[1]}, V^{[2]}, \dots$ denote the write operation executions to v , where

$$V^{[1]} \longrightarrow V^{[2]} \longrightarrow \dots$$

We let $v^{[i]}$ denote the value written by the operation execution $V^{[i]}$.

Thus $V^{[i]}$ is a *write* $v := v^{[i]}$ operation execution. We assume that the variable can be initialized to either possible value. (The initial value of a variable can be specified as part of the process's program.)

If a read is not concurrent with any write, then we expect it to obtain the value written by the most recent write—or the initial value if it precedes all writes. However, it turns out that we need a somewhat stronger requirement. To justify it, we return to our space-time view of operations. The value of a variable must be stored in some collection of objects. Communication is effected by the reads and writes acting on these objects—i.e., by each read and write operation execution containing events that lie on the world lines of these objects. A read or write operation may also contain “internal” events not on the world line of any of these shared objects. For example, if the variable is implemented by a flip-flop accessed by the reader and writer over separate wires, the flip-flop itself is the shared object and the events occurring on the wires are internal events. The internal events of a write do not directly affect a read. However, for a write to precede (\longrightarrow) a read, all events of the write, including internal events, must precede all events of the read.

For two operation executions A and B on the same variable, we say that A “effectively precedes” B if every event in A that lies on the world line of one of the shared objects precedes any events in B that lie on the same object's world line. For a read to obtain the value written by $V^{[k]}$, it suffices that (i) $V^{[k]}$ effectively precedes the read, and (ii) the read effectively precedes $V^{[k+1]}$. “Effectively precedes” is weaker than “precedes”, since it does not specify any ordering on internal events, so this condition is stronger than requiring that the read obtain the correct value if it is not concurrent with any write.

This definition of “effectively precedes” involves events, which are not part of our formalism, so we cannot define this exact concept. However, observe that if events a and b lie on the same world line, then either $a \longrightarrow b$ or $b \longrightarrow a$. Hence, if A and B both have events occurring on the same world line, then $A \dashrightarrow B$ and/or $B \dashrightarrow A$. If $B \dashv A$, then no event in B precedes any event in A . Hence, $A \dashrightarrow B$ and

$B \dashv A$ imply that A effectively precedes B . We therefore are led to the following definition:

Definition 1 We say that two operation executions A and B are *effectively nonconcurrent* if either $A \rightarrowtail B$ or $B \rightarrowtail A$, but not both.

If two operation executions are effectively nonconcurrent according to this definition, then one “effectively precedes” the other according to the above definition in terms of events. We therefore expect a read that is effectively nonconcurrent with every write to obtain the correct value. This leads us to the following requirement.

C2. A read R of v that is effectively nonconcurrent with every $V^{[i]}$ obtains the value $v^{[k]}$, where k is the largest number such that $V^{[k]} \rightarrowtail R$, or it obtains the initial value if there is no such k .

It follows from A2 and A5 that the set of k such that $V^{[k]} \rightarrowtail R$ is finite, so C2 specifies the value obtained by a read that is effectively nonconcurrent with every write. The only assumption we make about a read that is “effectively concurrent” with some write is that it obtain either the value *true* or the value *false* (by C1).

In the above space-time discussion of reading and writing, it is clear that for communication to take place, every pair of reads and writes must have events on the world line of the same object. The following requirement is therefore quite reasonable (although it may not be obvious why we need it).

C3. If R is a read of the communication variable v , then for every write $V^{[i]}$ of v :
 $R \rightarrowtail V^{[i]}$ or $V^{[i]} \rightarrowtail R$ (or both).

It has been argued that the kind of communication variable we are assuming is equivalent to one in which reads and writes are atomic actions that cannot be concurrent. The reasoning used is as follows.

If the value of the variable is not changed by a write, then there is no reason to do the write. We may therefore assume that a process executes a write only if it will change the value. By C1, a read that is concurrent with such a write must obtain either the old or the new value, since those are the only possible values. If the read obtains the old value, then we may consider it to have preceded the write, and if it obtains the new value then we may consider the write to have preceded it.⁴

4. In fact, we made this unfortunate claim in our original correctness proof for the bakery algorithm [5]. Happily, it was only the proof and not the algorithm that turned out to be incorrect.

This reasoning is fallacious under our assumptions because if two successive reads are concurrent with the same write, then the first read can obtain the new value and the second the old value. This is impossible if reads and writes are nonconcurrent atomic actions.

Several people have devised mutual exclusion algorithms using communication variables similar to ours, except with the stronger assumption that writing and reading are atomic operations [13,14]. We believe that these algorithms do not work with the more primitive type of communication variable that we are assuming.⁵ Other than the ones mentioned here, we know of no published mutual exclusion algorithms that are correct using these communication variables.

4 Processes

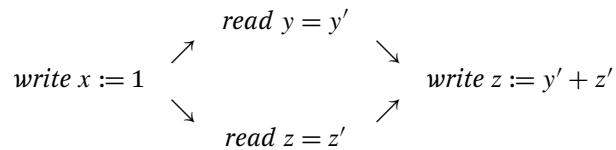
An algorithm implements higher-level operations such as *request service* in terms of lower-level ones like reading and writing a one-bit variable. A synchronization problem is posed as a set of conditions on the higher-level system execution—for example, that each *request service* operation execution is followed by a *grant service* operation execution. A solution consists of a specification of a lower-level system execution together with a higher-level view—for example, an algorithm for generating reads and writes together with a specification of which sets of these lower-level operation executions correspond to *request service* and *grant service* executions. The system execution defined by this higher-level view must satisfy the problem conditions.

We now consider how the lower-level system execution is specified. We assume that the set of all elementary operation executions is partitioned into N sets called *processes*. A process is described by an ordinary program, each operation execution of the process being generated by the execution of some statement in its program. For example, suppose the program for a process contains the following program statement:

```
begin
  x := 1;
  z := y + z
end
```

5. We have found counterexamples to the simpler algorithms, and have no reason to expect the more complicated ones to work better.

Executing this statement might generate the following four elementary operation executions, with the indicated \rightarrow relations.



Although we think of the program as generating the operation executions, formally the set of operation executions is given and the processes' programs provide a set of conditions on it. For example, if this statement were the only place where y is read, then it would provide the following formal condition:

For every $\text{read } y = y'$ operation execution, there must exist three operation executions $\text{write } x := 1$, $\text{read } z = z'$, and $\text{write } z := y' + z'$ such that the above \rightarrow relations hold.

Each process will be described by a program written in an Algol-like language with two kinds of program variables:

- *Private variables* read and written by that process only.
- *Communication variables* used for interprocess communication.

We can define a formal semantics for the programming language as follows. The elementary operation executions of a process are of the form $\text{write } v := v'$ or $\text{read } v = v'$, where v is a variable and v' is an element in the range of values of that variable. The variable v must be one that the process can write or read, respectively. For the critical section problem, there are also elementary operation executions of the form *critical section execution* and *noncritical section execution*.

We assume that C0–C3 hold for reads and writes of communication variables. We also assume that C0 and C2 hold for private variables. We will not need C1 or C3 because a read of a private variable will never be concurrent with a write of that variable. In fact, a read will not be concurrent with any write performed by the same process.

We now indicate how a process's program can be formally translated into a set of conditions on possible system executions. Syntactically, a program is composed of a hierarchy of program statements—more complicated statements being built up from simpler ones.⁶ In any system execution, to each program statement cor-

6. If function calls are permitted, then we have to include expression evaluations as well as statement executions in this hierarchy.

responds a (not necessarily elementary) operation execution—intuitively, it is the set of elementary operation executions performed when executing that statement. The execution of the entire program, which is a single statement, is a single operation execution consisting of all the process's elementary operation executions. The semantics of each type of statement in the language is defined by a collection of axioms on the (set of elementary operation executions in the) execution of a statement of that type. For assignment statements, we have the following axiom:

An execution of the statement

$$v := F(v_1, \dots, v_m)$$

consists of the elementary operation executions $\text{read } v_1 = v'_1, \dots, \text{read } v_m = v'_m$, and $\text{write } v := F(v'_1, \dots, v'_m)$, where each read precedes (\rightarrow) the write.

This axiom, together with conditions C0–C3 for communication variables and C0 and C2 for private variables, defines the semantics of the simple assignment statement.

The following axioms define the semantics of the concatenation construction $S; T$. (Recall that a statement execution, being a set of elementary operation executions, is defined to terminate if and only if it consists of a finite number of terminating operation executions.)

An execution of $S; T$ is one of the following:

- A nonterminating execution of S .
- An operation execution of the form $A \cup B$, where
 - A is a terminating execution of S .
 - B is an execution of T .
 - $A \rightarrow B$.

In this way, one can give a complete formal semantics for our programming language. However, we will not bother to do so, and will reason somewhat informally about system executions. We merely note the following properties:

- A write is not concurrent with any other operation generated by the same process. (However, it may be concurrent with operations generated by other processes.)
- Any elementary operation execution is concurrent with only a bounded number of elementary operation executions in the same process.

5 Multiple-Reader Variables

Thus far, we have assumed that communication variables can be read by only a single process. Using such variables, it is easy to construct a communication variable satisfying C0–C3 that can be read by several processes, though only written by one process. To implement a communication variable v that can be written by process i and read by processes $1, \dots, N$, we use an array $v[1], \dots, v[N]$ of variables, where $v[j]$ can be written by i and read by j . (All the $v[j]$ are communication variables except for $v[i]$, which is a private variable of process i .) Any statement $v := \dots$ in process i 's program is implemented as an operation of assigning the value of the right-hand expression to each element of the array, and any occurrence of the variable v in an expression within the program of process j is interpreted as an occurrence of $v[j]$. The fact that this construction works is implied by the following result, whose proof is left to the reader.

Theorem 1 For each $j \neq i$, let $v[j]$ be a communication variable that is written by process i and read by process j . Assume that for all j, j' and all k :

1. The initial values of $v[j]$ and $v[j']$ are equal;
2. $v[j]^{[k]} = v[j']^{[k]}$;
3. $V[j]^{[k]} \longrightarrow V[j']^{[k+1]}$.

Let the initial value of v be defined to equal the initial value of the $v[j]$, let $V^{[k]}$ be defined to be $\{V[1]^{[k]}, \dots, V[N]^{[k]}\}$, and define a read of v by a process $j \neq i$ to be a read of $v[j]$. Then C0–C3 are satisfied by the variable v (where \longrightarrow and \dashrightarrow are defined for the set of operation executions $V^{[k]}$ by (2.1)).

Formally, we are defining a higher-level view whose operation executions are the same as those of the original system execution except that the reads and writes of the $v[j]$ are partitioned into reads and writes of v . This theorem shows that the resulting higher-level system execution satisfies C0–C3. We take the reads and writes of v to be elementary operation executions, ignoring the lower-level operation executions that comprise them.

We will therefore assume that a communication variable can be written by its owner and read by any process. However, we must remember that the “cost” of implementing such a communication variable may depend upon the number of processes that actually read it. If the physical communication mechanism involves wires that join two processors, then the number of wires needed to implement a communication variable equals the number of readers of that variable, so a variable read by r processes may be almost r times as expensive as one read by a single process. However, it is quite reasonable to suppose that the variable could be

implemented with a single wire to which each reader is connected. In this case, the cost of an r -reader variable may not be much greater than the cost of a single-reader variable.

6 Discussion of the Assumptions

In this section, we have made some tacit assumptions that may have passed unnoticed. The most obvious of these is the assumption that each process knows in advance who it might communicate with. This assumption seems to us to be reasonable for an underlying physical model in which processors (the physical hardware that executes processes) are connected in pairs by direct physical connections—e.g., wires or optical fibers. In such a model, it is natural to assume that a processor knows the existence of every physical connection. Indeed, it is only for such a model that a communication variable owned by a single process is reasonable. Thus, our work is not applicable to systems of anonymous processors connected along a common wire, as in an Ethernet [10].

Our first two assumptions, C0 and C1, appear quite innocent. However, the fact that reading and writing are not synchronized means that the reader can become suspended for arbitrarily long in a meta-stable state if it happens to read at exactly the wrong time. This is the “arbiter problem” discussed in [2] and [12]. As explained in [2], one can construct a device in which the reader has probability zero of remaining in such a meta-stable state forever. Hence, our assumptions can be satisfied if we interpret truth to mean “true with probability one”.

There is an additional subtle assumption hidden in the combination of C2 with the ordering of operation executions within a process that we have been assuming. Suppose $v := \text{true}; \dots$ is part of the program for process i . We are assuming that the $\text{write } v := \text{true}$ generated by an execution of the first statement precedes any operation execution A generated by the subsequent execution of the next statement. Now suppose that this is the last $\text{write } v$ generated by process i , and that there is a $\text{read } v$ execution by another process that is preceded by A . By A1, the $\text{read } v$ is preceded by the $\text{write } v := \text{true}$, and since this is the last $\text{write } v$ operation execution, C2 implies that the $\text{read } v$ must obtain the value true .

Let us consider what this implies for an implementation. To guarantee that the $\text{read } v$ obtains the value true , after executing the $\text{write } v := \text{true}$ the writer must be sure that v has settled into a stable state before beginning the next operation. For example, if the value of v is represented by the voltage level on the wire joining two processors, then the writer cannot proceed until the new voltage has propagated to the end of the wire. If a bound cannot be placed upon the propagation time, then

such a simple representation cannot be used. Instead, the value must be stored in a flip-flop local to the writer, and the reader must interrogate its value by sending a signal along the wire and waiting for a response. Since the flip-flop is located at the writing process, and is set only by that process, it is reasonable to assume a bound upon its settling time. The wire, with its unknown delay, becomes part of the reading process. Thus, although satisfying our assumption in a distributed process poses difficulties, they do not seem to be insurmountable. In any case, we know of no way to achieve interprocess synchronization without such an assumption.

7 Conclusion

In Section 2, we developed a formalism for reasoning about concurrent systems that does not assume the existence of atomic operations. This formalism has been further developed in [7], which addresses the question of what it means for a lower-level system to implement a higher-level one.

Section 3 considered the nature of interprocess communication, and argued that the simplest, most primitive form of communication that can be used to solve the mutual exclusion problem consists of a very weak form of shared register that can be written and read concurrently. Interprocess communication is considered in more detail in [8], where the form of shared register we have defined is called a *safe* register. Algorithms for constructing stronger registers from safe ones are given in [8].

Acknowledgments

Many of these ideas have been maturing for quite a few years before appearing on paper for the first time here. They have been influenced by a number of people during that time, most notably Carel Scholten, Edsger Dijkstra, Chuck Seitz, Robert Keller, and Irene Greif.

References

- [1] BRINCH HANSEN, P. Concurrent programming concepts. *Comput. Surv.* 5 (1973) 223–245, 1973.
- [2] CHANEY, T. J. and MOLNAR, C. E. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.* C-22 (Apr. 1973), 421–422.
- [3] DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (Sept. 1965), 569.

- [4] EINSTEIN, A. Zur electrodynamik bewegter korper. *Ann. Physik*, 17 (1905). Translated as: On the electrodynamics of moving bodies. In *The Principle of Relativity*, Dover, New York, pp. 35–65.
- [5] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17, 8 (Aug. 1974), 453–455.
- [6] LAMPORT, L. A new approach to proving the correctness of multiprocess programs. *Prog. Lang. Syst.* 1, 1 (July 1979), 84–97.
- [7] LAMPORT, L. On interprocess communication—Part I: Basic formalism. *Dist. Comput.* (to appear).
- [8] LAMPORT, L. On interprocess communication—Part II: Algorithms. *Dist. Comput.* (to appear).
- [9] LAMPORT, L. The mutual exclusion problem: Part II—Statement and solutions. *J. ACM* 33, 2 (Apr. 1986), 327–348.
- [10] METCALFE, R. and BOGGS, D. R. Ethernet: distributed packet switching for local computer networks. *Commun. ACM* 19, 7 (July 1976), 395–404.
- [11] MINKOWSKI, H. Space and Time. In *The Principle of Relativity*. Dover, New York, pp. 73–91.
- [12] PALAIS, R. and LAMPORT, L. On the glitch phenomenon. Tech. Rep. CA-7611-0811, Massachusetts Computer Associates, Wakefield, Mass., Nov. 1976.
- [13] PETERSON, G. and FISCHER, M. J. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th ACM Symposium on Theory of Computing* (Boulder, Colo., May 2–4). ACM, New York, 1977, pp. 91–97.
- [14] RIVEST, R. L. and PRATT, V. R. The mutual exclusion problem for unreliable processes: Preliminary report. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. IEEE, New York, 1976, pp. 1–80.
- [15] SCHWARTZ, J. T. *Relativity in Illustrations*. New York University Press, New York, 1962.
- [16] TAYLOR, E. F. and WHEELER, J. A.. *Space-Time Physics*. W. H. Freeman, San Francisco, 1966.

The Mutual Exclusion Problem: Part II—Statement and Solutions

Leslie Lamport (Digital Equipment Corporation, Palo Alto, California)

Abstract

The theory developed in Part I is used to state the mutual exclusion problem and several additional fairness and failure-tolerance requirements. Four “distributed” N -process solutions are given, ranging from a solution requiring only one communication bit per process that permits individual starvation, to one requiring about $N!$ communication bits per process that satisfies every reasonable fairness and failure-tolerance requirement that we can conceive of.

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management—*concurrency; multiprocessing/multiprogramming; mutual exclusion; synchronization*; D.4.5 [Operating Systems]: Reliability—*fault-tolerance*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Critical section, shared variables

Most of this work was performed while the author was at SRI International, where it was supported in part by the National Science Foundation under grant number MCS-78-16783.

Author's address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1986 ACM 0004-5411/86/0400-0327 \$00.75

Paper originally published in *Journal of the Association for Computing Machinery*, 33(2), April 1986, pp. 327–348.

1 Introduction

This is the second part of a two-part paper on the mutual exclusion problem. In Part I [15], we described a formal model of concurrent systems and used it to define a primitive interprocess communication mechanism (communication variables) that assumes no underlying mutual exclusion. In this part, we consider the mutual exclusion problem itself.

The mutual exclusion problem was first described and solved by Dijkstra in [2]. In this problem, there is a collection of asynchronous processes, each alternately executing a critical and a noncritical section, that must be synchronized so that no two processes ever execute their critical sections concurrently. Dijkstra's original solution was followed by a succession of others, starting with [6].

These solutions were motivated by practical concerns—namely, the need to synchronize multiprocess systems using the primitive operations provided by the hardware. More recent computers usually provide sophisticated synchronization primitives that make it easy to achieve mutual exclusion, so these solutions are of less practical interest today. However, mutual exclusion lies at the heart of most concurrent process synchronization, and the mutual exclusion problem is still of great theoretical significance. This paper carefully examines the problem and presents new solutions of theoretical interest. Although some of them may be of practical value as well—especially in distributed systems—we do not concern ourselves here with practicality.

All of the early solutions assumed a central memory, accessible by all processes, which was typical of the hardware in use at the time. Implementing such a central memory requires some mechanism for guaranteeing mutually exclusive access to the individual memory cells by the different processes. Hence, these solutions assume a lower-level “hardware” solution to the very problem they are solving. From a theoretical standpoint, they are thus quite unsatisfactory as solutions to the mutual exclusion problem. The first solution that did not assume any underlying mutual exclusion was given in [7]. However, it required an unbounded amount of storage, so it too was not theoretically satisfying. The only other published solution we are aware of that does not assume mutually exclusive access to a shared resource is by Peterson [17].

Here, in Part II, we present four solutions that do not assume any underlying mutual exclusion, using the concurrently accessible registers defined in Part I [15]. They are increasingly stronger, in that they satisfy stronger conditions, and more expensive, in that they require more storage. The precise formulation of the mutual exclusion problem and of the various fairness and failure-tolerance assumptions, is based upon the formalism of Part I.

2**The Problem**

We now formally state the mutual exclusion problem, including a number of different requirements that one might place upon a solution. We exclude from consideration only the following types of requirements:

- efficiency requirements involving space and time complexity;
- probabilistic requirements, stating that the algorithm need only work with probability one (solutions with this kind of requirement have recently been studied by Rabin [19]);
- generalizations of the mutual exclusion problem, such as allowing more than one process in the critical section at once under certain conditions [4, 8], or giving the processes different priorities [8].

Except for these exclusions and one other omission (r -bounded waiting) mentioned below, we have included every requirement we could think of that one might reasonably want to place upon a solution.

2.1 Basic Requirements

We assume that each process's program contains a noncritical section statement and a critical section statement, which are executed alternately. These statements generate the following sequence of elementary operation executions in process i :

$$NCS_i^{[1]} \longrightarrow CS_i^{[1]} \longrightarrow NCS_i^{[2]} \longrightarrow CS_i^{[2]} \longrightarrow \dots$$

where $NCS_i^{[k]}$ denotes the k th execution of process i 's noncritical section, $CS_i^{[k]}$ denotes the k th execution of its critical section, and \longrightarrow is the precedence relation introduced in Part I. Taking $NCS_i^{[k]}$ and $CS_i^{[k]}$ to be elementary operation executions simply means that we do not assume any knowledge of their internal structure, and does not imply that they are of short duration.

We assume that the $CS_i^{[k]}$ are terminating operation executions, which means that process i never “halts” in its critical section. However, $NCS_i^{[k]}$ may be nonterminating for some k , meaning that process i may halt in its noncritical section.

The most basic requirement for a solution is that it satisfy the following:

Mutual Exclusion Property. For any pair of distinct processes i and j , no pair of operation executions $CS_i^{[k]}$ and $CS_j^{[k']}$ are concurrent.

In order to implement mutual exclusion, we must add some synchronization operations to each process's program. We make the following requirement on these additional operations.

No other operation execution of a process can be concurrent with that process's critical or noncritical section operation executions.

This requirement was implicit in Dijkstra's original statement of the problem, but has apparently never been stated explicitly before.

The above requirement implies that each process's program may be written as follows:

```
initial declaration;
repeat forever
  noncritical section;
  trying;
  critical section;
  exit;
end repeat
```

The *trying* statement is what generates all the operation executions between a noncritical section execution and the subsequent critical section execution, and the *exit* statement generates all the operation executions between a critical section execution and the subsequent noncritical section execution. The *initial declaration* describes the initial values of the variables. A solution consists of a specification of the *initial declaration*, *trying* and *exit* statements.

A process i therefore generates the following sequence of operation executions:

$$NCS_i^{[1]} \rightarrow \text{trying}_i^{[1]} \rightarrow CS_i^{[1]} \rightarrow \text{exit}_i^{[1]} \rightarrow NCS_i^{[2]} \rightarrow \dots$$

where $\text{trying}_i^{[1]}$ denotes the operation execution generated by the first execution of the *trying* statement, etc.

The second basic property that we require of a solution is that there be no deadlock. Deadlock occurs when one or more processes are “trying to enter” their critical sections, but no process ever does. To say that a process tries forever to enter its critical section means that it is performing a nonterminating execution of its *trying* statement. Since every critical section execution terminates, the absence of deadlock should mean that if some process's *trying* statement doesn't terminate, then other processes must be continually executing their critical sections. However, there is also the possibility that a deadlock occurs because all the processes are stuck in their *exit* statements. The possibility of a nonterminating *exit* execution complicates the statement of the properties and is of no interest here, since the *exit* statements in all our algorithms consist of a fixed number of terminating op-

erations. We will therefore simply require of an algorithm that every *exit* execution terminates.

The absence of deadlock can now be expressed formally as follows:

Deadlock Freedom Property. If there exists a nonterminating *trying* operation execution, then there exist an infinite number of critical section operation executions.

These two properties, mutual exclusion and deadlock freedom, were the requirements for a mutual exclusion solution originally stated by Dijkstra in [2]. (Of course, he allowed mutually exclusive access to a shared variable in the solution.) They are the minimal requirements one might place on a solution.

2.2 Fairness Requirements

Deadlock freedom means that the entire system of processes can always continue to make progress. However, it does not preclude the possibility that some individual process may wait forever in its *trying* statement. The requirement that this cannot happen is expressed by:

Lockout Freedom Property. Every *trying* operation execution must terminate.

This requirement was first stated and satisfied by Knuth in [6].

Lockout freedom means that any process i trying to enter its critical section will eventually do so, but it does not guarantee when. In particular, it allows other processes to execute their critical sections arbitrarily many times before process i executes its critical section. We can strengthen the lockout freedom property by placing some kind of fairness condition on the order in which trying processes are allowed to execute their critical sections.

The strongest imaginable fairness condition is that if process i starts to execute its *trying* statement before process j does, then i must execute its critical section before j does. Such a condition is not expressible in our formalism because “starting to execute” is an instantaneous event, and such events are not part of the formalism. However, even if we were to allow atomic operations—including atomic reads and writes of communication variables—so our operations were actually instantaneous events, one can show that this condition cannot be satisfied by any algorithm. The reason is that with a single operation, a process can either tell the other processes that it is in its *trying* statement (by performing a write) or else check if other processes are in their *trying* statements (by performing a read), but not both. Hence, if two processes enter their *trying* statements at very nearly the same time, then there

will be no way for them to decide which one entered first. This result can be proved formally, but we will not bother to do so.

The strongest fairness condition that can be satisfied is the following *first-come-first-served* (FCFS) condition. We assume that the *trying* statement consists of two substatements—a *doorway* whose execution requires only a bounded number of elementary operation executions (and hence always terminates), followed by a *waiting* statement. We can require that, if process i finishes executing its *doorway* statement before process j begins executing its *doorway* statement, then i must execute its critical section before j does. Letting $doorway_i^{[k]}$ and $waiting_i^{[k]}$ denote the k th execution of the *doorway* and *waiting* statements by process i , this condition can be expressed formally as follows:

First-Come, First-Served Property. For any pair of processes i and j and any execution $CS_j^{[m]}$: if $doorway_i^{[k]} \rightarrow doorway_j^{[m]}$, then $CS_i^{[k]} \rightarrow CS_j^{[m]}$.

(The conclusion means that $CS_i^{[k]}$ is actually executed.)

The FCFS property states that processes will not execute their critical sections “out of turn”. However, it does not imply that any process ever actually executes its critical section. In particular, FCFS does not imply deadlock freedom. However, FCFS and deadlock freedom imply lockout freedom, as we now show.

Theorem 1 FCFS and deadlock freedom imply lockout freedom.

Proof Suppose $trying_i^{[k]}$ is nonterminating. Since there are a finite number of processes, the deadlock freedom property implies that some process j performs an infinite number of $CS_j^{[m]}$ executions, and therefore an infinite number of $doorway_j^{[m]}$ executions. It then follows from Axiom A5 of Part I that $doorway_i^{[k]} \rightarrow doorway_j^{[m]}$ for some m . The FCFS property then implies the required contradiction. ■

The requirement that executing the *doorway* take only a bounded number of elementary operation executions means that a process does not have to wait inside its *doorway* statement. Formally, the requirement is that there be some *a priori* bound—the same bound for any possible execution of the algorithm—on the number of elementary operation executions in each $doorway_i^{[k]}$. Had we only assumed that the *doorway* executions always terminate, then any lockout-free solution is always FCFS, where the *doorway* is defined to be essentially the entire *trying* statement. This requirement seems to capture the intuitive meaning of “first-come, first-served”. A weaker notion of FCFS was introduced in [18], where it was only required that a process in its *doorway* should not have to wait for a process in its critical or noncritical section. However, we find that definition rather arbitrary.

Michael Fischer has also observed that a FCFS algorithm should not force a process to wait in its *exit* statement. Once a process has finished executing its critical section, it may execute a very short noncritical section and immediately enter its *trying* statement. In this case, the *exit* statement is effectively part of the next execution of the *doorway*, so it should involve no waiting. Hence, any $exit_i^{[k]}$ execution should consist of only a bounded number of elementary operation executions for a FCFS solution. As we mentioned above, this is true of all the solutions described here.

An additional fairness property intermediate between lockout freedom and FCFS, called *r-bounded waiting*, has also been proposed [20]. It states that after process i has executed its *doorway*, any other process can enter its critical section at most r times before i does. Its formal statement is the same as the above statement of the FCFS property, except with $CS_j^{[m]}$ replaced by $CS_j^{[m+r]}$.

2.3 Premature Termination

Thus far, all our properties have been constraints upon what the processes may do. We now state some properties that give processes the freedom to behave in certain ways not explicitly indicated by their programs. We have already required one such property by allowing nonterminating executions of the noncritical section—i.e., we give the process the freedom to halt in its noncritical section. It is this requirement that distinguishes the mutual exclusion problem from a large class of synchronization problems known as “producer/consumer” problems [1]. For example, it prohibits solutions in which processes must take turns entering their critical section.

We now consider two kinds of behavior in which a process can return to its noncritical section from any arbitrary point in its program. In the first, a process stops the execution of its algorithm by setting its communication variables to certain default values and halting. Formally, this means that anywhere in its algorithm, a process may execute the following operation:

```
begin
  set all communication variables to their default values;
  halt
end
```

For convenience, we consider the final halting operation execution to be a nonterminating noncritical section execution. The default values are specified as part of the algorithm. For all our algorithms, the default value of every communication variable is the same as its initial value.

This type of behavior has been called “failure” in previous papers on the mutual exclusion problem. However, we reserve the term “failure” for a more insidious kind of behavior, and call the above behavior *shutdown*. If the algorithm satisfies a property under this type of behavior, then it is said to be *shutdown safe* for that property.

Shutdown could represent the physical situation of “unplugging” a processor. Whenever a processor discovers that another processor is unplugged, it does not try to actually read that processor’s variables, but instead uses their default values. We require that the processor never be “plugged back in” after it has been unplugged. We show below that this is really equivalent to requiring that the processor remain unplugged for a sufficiently long time.

The second kind of behavior is one in which a process deliberately *aborts* the execution of its algorithm. Abortion is the same as shutdown except for three things:

- The process returns to its noncritical section instead of halting.
- Some of its communication variables are left unchanged. (Which ones are specified as part of the algorithm.)
- A communication variable is not set to its default value if it already has that value.¹

Formally, an abortion is an operation execution consisting of a collection of writes that set certain of the process’s communication variables to their default values, followed by (→) a noncritical section execution. (The noncritical section execution may then be followed by a *trying* statement execution—or by another abortion.) For our algorithms, the value of a communication variable is set by an abortion if there is an explicitly declared initial value for the variable, otherwise it is left unchanged by the abortion. If an algorithm satisfies a property with this type of behavior, then it is said to be *abortion safe* for that property.

2.4 Failure

Shutdown and abortion describe fairly reasonable kinds of behavior. We now consider unreasonable kinds of behavior, such as might occur in the event of process failure. There are two kinds of faulty behavior that a failed process could exhibit.

1. Remember that setting a variable to its old value is not a “no-op”, since a read that is concurrent with that operation may get the wrong value. If communication variables were set every time the process aborted, repeated abortions would be indistinguishable from the “malfunctioning” behavior considered below.

- unannounced death*, in which it halts undetectably;
- malfunctioning*, in which it keeps setting its state, including the values of its communication variables, to arbitrary values.

An algorithm that can handle the first type of faulty behavior must use real-time clocks, otherwise there is no way to distinguish between a process that has died and one that is simply pausing for a long time between execution steps. An example of an algorithm (not a solution to our mutual exclusion problem) that works in the presence of such faulty behavior can be found in [10]. Consideration of this kind of behavior is beyond the scope of this paper.

A malfunctioning process obviously cannot be prevented from executing its critical section while another process's critical section execution is in progress. However, we may still want to guarantee mutual exclusion among the nonfaulty processes. We therefore assume that a malfunctioning process does not execute its critical section. (A malfunctioning process that executes its *critical section* code is simply defined not to be executing its critical section.)

A malfunctioning process can also disrupt things by preventing nonfaulty processes from entering their critical sections. This is unavoidable, since a process that malfunctions after entering its critical section could leave its communication variables in a state indicating that it is still in the critical section. What we can hope to guarantee is that if the process stops malfunctioning, then the algorithm will resume its normal operation. This leaves two types of behavior to be considered, which differ in how a process stops malfunctioning.

The first type of failure allows a failed process to execute the following sequence of actions.

- It malfunctions for a while, arbitrarily changing the values of its communication variables.
- It then aborts—setting all its communication variables to some default values.
- It then resumes normal behavior, never again malfunctioning.

This behavior represents a situation in which a process fails, its failure is eventually detected and it is shut down, and the process is repaired and restored to service. The assumption that it never again malfunctions is discussed below.

Formally, this means that each process may perform at most one operation execution composed of the following sequence of executions (ordered by the \rightarrow relation):

- a *malfunction* execution, consisting of a finite collection of writes to its communication variables.
- a collection of writes that sets each communication variable to its default value.
- a noncritical section execution.

The above operation execution will be called a *failure*. If a property of a solution remains satisfied under this kind of behavior, then the solution is said to be *fail-safe* for that property. Note that we do not assume failure to be detectable; one process cannot tell that another has failed (unless it can infer from the values of the other process's variables that a failure must have occurred).

The second type of failure we consider is one in which a process malfunctions, but eventually stops malfunctioning and resumes forever its normal behavior, starting in any arbitrary state. This behavior represents a transient fault.

If such a failure occurs, we cannot expect the system immediately to resume its normal operation. For example, the malfunctioning process might resume its normal operation just at the point where it is about to enter its critical section—while another process is executing its critical section. The most we can require is that after the process stops malfunctioning, the system *eventually* resumes its correct operation.

Since we are interested in the eventual operation, we need only consider what happens after every process has stopped malfunctioning. The state of the system at that time can be duplicated by starting all processes at arbitrary points in their program, with their variables having arbitrary values. In other words, we need only consider the behavior obtained by having each process do the following:

- execute a *malfunction* operation;
- then begin normal execution at any point in its program.

This kind of behavior will be called a *transient malfunction*. Any operation execution that is not part of the malfunction execution will be called a *normal* operation execution.

Unfortunately, deadlock freedom and lockout freedom cannot be achieved under this kind of transient malfunction behavior without a further assumption. To see why, suppose a malfunctioning process sets its communication variables to the values they should have while executing its critical section, and then begins normal execution with a nonterminating noncritical section execution. The process will al-

ways appear to the rest of the system as if it is executing its critical section, so no other process can ever execute its critical section.

To handle this kind of behavior, we must assume that a process executing its noncritical section will eventually set its communication variables to their default values. Therefore, we assume that instead of being elementary, the noncritical section executions are generated by the following program:

```
while ?
  do abort ;
    noncritical operation od
```

where the “?” denotes some unknown condition, which could cause the **while** to be executed forever, and every execution of the *noncritical operation* terminates. Recall that an *abort* execution sets certain communication variables to their default values if they are not already set to those values.

We now consider what it means for a property to hold “eventually”. Intuitively, by “eventually” we mean “after some bounded period of time” following all the malfunctions. However, we have not introduced any concept of physical time. The only unit of time implicit in our formalism is the time needed to perform an operation execution. Therefore, we must define “after some bounded period of time” to mean “after some bounded number of operation executions”. The definition we need is the following.

Definition 1 A *system step* is an operation execution consisting of one normal elementary operation execution from every process. An operation execution A is said to occur *after t system steps* if there exist system steps S_1, \dots, S_t such that $S_1 \rightarrow \dots \rightarrow S_t \rightarrow A$.

It is interesting to note that we could introduce a notion of time by defining the “time” at which an operation occurs to be the maximum t such that the operation occurs after t system steps (or 0 if there is no such t). Axioms A5 and A6 of Part I imply that this maximum always exists. Axiom A5 and the assumption that there are no nonterminating elementary operation executions imply that “time” increases without bound—i.e., there are operations occurring at arbitrarily large “times”. Since we only need the concept of eventuality, we will not consider this way of defining “time”.

We can now define what it means for a property to hold “eventually”. Deadlock freedom and lockout freedom state that something eventually happens—for example, deadlock freedom states that so long as some process is executing its *trying* operation, then some process eventually executes its critical section. Since “eventually X eventually happens” is equivalent to “ X eventually happens”, requiring

that these two properties eventually hold is the same as simply requiring that they hold.

We say that the mutual exclusion and FCFS properties eventually hold if they can be violated only for a bounded “length of time”. Thus, the mutual exclusion property eventually holds if there is some t such that any two critical section executions $CS_i^{[k]}$ and $CS_j^{[m]}$ that both occur after t system steps are not concurrent. Similarly, the FCFS property holds eventually if it holds whenever both the *doorway* executions occur after t system steps. The value of t must be independent of the particular execution of the algorithm, but it may depend upon the number N of processes.

If a property eventually holds under the above type of transient malfunction behavior, then we say that the algorithm is *self-stabilizing* for that property. The concept of self-stabilization is due to Dijkstra [3].

Remarks on “Forever”. In our definition of failure, we could not allow a malfunctioning process to fail again after it had resumed its normal behavior, since repeated malfunctioning and recovery can be indistinguishable from continuous malfunctioning. However, if an algorithm satisfies any of our properties under the assumption that a process may malfunction only once, then it will also satisfy the property under repeated malfunctioning and recovery—so long as the process waits long enough before malfunctioning again.

The reason for this is that all our properties require that something either be true at all times (mutual exclusion, FCFS) or that something happen in the future (deadlock freedom, lockout freedom). If something remains true during a malfunction, then it will also remain true under repeated malfunctioning. If something must happen eventually, then because there is no “crystal ball” operation that can tell if a process will abort in the future,² another malfunction can occur after the required action has taken place. Therefore, an algorithm that is fail-safe for such a property must also satisfy the property under repeated failure, if a failed process waits long enough before executing its *trying* statement again. Similar remarks apply to shutdown and transient malfunction.

3 The Solutions

We now present four solutions to the mutual exclusion problem. Each one is stronger than the preceding one in the sense that it satisfies more properties, and is more expensive in that it requires more communication variables.

2. Such operations lead to logical contradictions—for example, if one process executes “set x true if process i will abort”, and process i executes “abort if x is never set true”.

3.1 The Mutual Exclusion Protocol

We first describe the fundamental method for achieving mutual exclusion upon which all the solutions are based. Each process has a communication variable that acts as a synchronizing “flag”. Mutual exclusion is guaranteed by the following protocol: in order to enter its critical section, a process must first set its flag true and then find every other process’s flag to be false. The following result shows that this protocol does indeed ensure mutual exclusion, where v and w are communication variables, as defined in Part I, that represent the flags of two processes, and A and B represent executions of those processes’ critical sections.

Theorem 2 Let v and w be communication variables, and suppose that for some operation executions A and B and some k and m :

- $V^{[k]} \rightarrow \text{read } w = \text{false} \rightarrow A$.
- $W^{[m]} \rightarrow \text{read } v = \text{false} \rightarrow B$.
- $v^{[k]} = w^{[m]} = \text{true}$.
- If $V^{[k+1]}$ exists then $A \rightarrow V^{[k+1]}$.
- If $W^{[m+1]}$ exists then $B \rightarrow W^{[m+1]}$.

Then A and B are not concurrent.

We first prove the following result, which will be used in the proof of the theorem. Its statement and proof use the formalism developed in Part I.

Lemma 1 Let v be a communication variable and R a $\text{read } v = \text{false}$ operation such that:

1. $v^{[k]} = \text{true}$
2. $V^{[k]} \rightarrow R$
3. $R \not\rightarrow V^{[k]}$

Then $V^{[k+1]}$ must exist and $V^{[k+1]} \rightarrow R$.

Proof Intuitively, the assumptions mean that $V^{[k]}$ “effectively precedes” R , so R cannot see any value written by a write that precedes $V^{[k]}$. Since R does not obtain the value written by $V^{[k]}$, it must be causally affected by a later write operation $V^{[k+1]}$. We now formalize this reasoning.

By A3 and the assumption that the writes of v are totally ordered, hypothesis 2 implies that $V^{[i]} \rightarrow R$ for all $i \leq k$. If $R \rightarrow V^{[i]}$ for some $i < k$, then A3 would imply $R \rightarrow V^{[k]}$, contrary to hypothesis 3. Hence, we conclude that R is effectively nonconcurrent with $V^{[i]}$ for all $i \leq k$. If $V^{[k]}$ were the last write to v , hypothesis 2 and C2 would imply that R has to obtain the value *true*, contrary to hypothesis 1. Therefore, the operation $V^{[k+1]}$ must exist.

We now prove by contradiction that $V^{[k+1]} \rightarrow R$. Suppose to the contrary that $V^{[k+1]} \not\rightarrow R$. C3 then implies that $R \rightarrow V^{[k+1]}$, which by A3 implies $R \rightarrow V^{[i]}$ for all $i \geq k+1$. A3 and the assumption that $V^{[k+1]} \not\rightarrow R$, implies that $V^{[i]} \not\rightarrow R$ for all $i \geq k+1$. Since we have already concluded that $V^{[i]} \rightarrow R$ for all $i \leq k$, C2 implies that R must obtain the value *true*, which contradicts hypothesis 1. This completes the proof that $V^{[k+1]} \rightarrow R$. ■

Proof of Theorem By C3, we have the following two possibilities:

1. $\text{read } w = \text{false} \rightarrow W^{[m]}$.
2. $W^{[m]} \rightarrow \text{read } w = \text{false}$.

(These are not disjoint possibilities.) We consider case (1) first. Combining (1) with the first two hypotheses of the theorem, we have

$$V^{[k]} \rightarrow \text{read } w = \text{false} \rightarrow W^{[m]} \rightarrow \text{read } v = \text{false}$$

By A4, this implies $V^{[k]} \rightarrow \text{read } v = \text{false}$. A2 and the lemma then imply that $V^{[k+1]}$ exists and $V^{[k+1]} \rightarrow \text{read } v = \text{false}$. Combining this with the fourth and second hypotheses gives

$$A \rightarrow V^{[k+1]} \rightarrow \text{read } v = \text{false} \rightarrow B$$

By A4, this implies $A \rightarrow B$, completing the proof in case (1).

We now consider case (2). Having already proved the theorem in case (1), we can make the additional assumption that case (1) does not hold, so $\text{read } w = \text{false} \not\rightarrow W^{[m]}$. We can then apply the lemma (substituting w for v and m for k) to conclude that $W^{[m+1]}$ exists and $W^{[m+1]} \rightarrow \text{read } w = \text{false}$. Combining this with the first and last hypotheses gives

$$B \rightarrow W^{[m+1]} \rightarrow \text{read } w = \text{false} \rightarrow A$$

A4 now implies $B \rightarrow A$, proving the theorem for this case. ■

We have written the proof of this theorem in full detail to show how A1–A4 and C0–C3 are used. In the remaining proofs, we will be more terse, leaving many of the details to the reader.

3.2 The One-Bit Solution

We now use the above protocol to obtain a mutual exclusion solution that requires only the single (one-bit) communication variable x_i for each process i . Obviously, no solution can work with fewer communication variables. This solution was also

```

private variable:  $j$  with range  $1 \dots N$ ;  

communication variable:  $x_i$  initially false;  

repeat forever  

  noncritical section;  

   $l: x_i := \text{true}$ ;  

  for  $j := 1$  until  $i - 1$   

    do if  $x_j$  then  $x_i := \text{false}$ ;  

    while  $x_j$  do od;  

    goto  $l$   

  fi  

  od;  

  for  $j := i + 1$  until  $N$   

    do while  $x_j$  do od od;  

    critical section;  

     $x_i := \text{false}$   

end repeat

```

Figure 1 The one-bit algorithm: Process i .

discovered independently by Burns [1a]. The algorithm for process i is shown in Figure 1, and its correctness properties are given by the following result.

Theorem 3 The One-Bit Algorithm satisfies the mutual exclusion and deadlock freedom properties, and is shutdown safe and fail-safe for these properties.

Proof To prove the mutual exclusion property, we observe that the above protocol is followed by the processes. More precisely, the mutual exclusion property is proved using Theorem 2, substituting x_i for v , x_j for w , $CS_i^{[k]}$ for A and $CS_j^{[k']}$ for B . This protocol is followed even under shutdown and failure behavior, so the algorithm is shutdown safe and fail-safe for mutual exclusion. To prove deadlock freedom, we first prove the following lemma. ■

Lemma 2 Any execution of the second **for** loop must terminate, even under shutdown and failure behavior.

Proof The proof is by contradiction. Let i be any process that executes a nonterminating second **for** loop. Then before entering the loop, i performs a finite number of *write* x_i executions, with the final one setting x_i true. We now prove by contradiction that every other process can also execute only a finite number of writes to its communication variable. Let k be the lowest-numbered process that performs an

infinite number of *write* x_k executions. Process k executes statement l infinitely many times. Since every lower-numbered process j executes only a finite number of writes to x_j , A5, A2 and C2 imply that all but a finite number of reads of x_j by k must obtain its final value. For k to execute statement l infinitely many times (and not get trapped during an execution of the first **for** loop's **while** statement), this final value must be *false* for every $j < k$. This implies that k can execute its first **for** loop only finitely many times before it enters its second **for** loop. But since the final value of x_i is *true*, this means that $k < i$, and that k can execute its second **for** loop only finitely many times before being trapped forever in the “**while** x_i ” statement in its second **for** loop. This contradicts the assumption that k performs an infinite number of *write* x_k executions.

We have thus proved that if the execution of the second **for** loop of any process is nonterminating, then every process can execute only a finite number of writes to its communication variable. The final value of a process's communication variable can be *true* only if the process executes its second **for** loop forever. Letting i be the highest-numbered process executing a nonterminating second **for** loop, so the final value of x_j is *false* for every $j > i$, we easily see that i must eventually exit this **for** loop, providing the required contradiction. Hence, every execution of the second **for** loop must eventually terminate. ■

Proof of Theorem (continued). We now complete the proof of the theorem by showing that the One-Bit Algorithm is deadlock free. Assume that some process performs a nonterminating *trying* execution. Let i be the lowest numbered process that does not execute a nonterminating noncritical section. (There is at least one such process—the one performing the nonterminating *trying* execution.) Each lower numbered process j performs a nonterminating noncritical section execution after setting its communication variable *false*. (This is true for shutdown and failure behavior too.) It follows from A5, A2, and C2 that if i performs an infinite number of reads of the variable x_j , then all but a finite number of them must return the value *false*. This implies that every execution of the first **for** loop of process i must terminate. But, by the above lemma, every execution of its second **for** loop must also terminate. Since we have assumed that every execution of its noncritical section terminates, this implies that process i performs an infinite number of critical section executions, completing the proof of the theorem. ■

The One-Bit Algorithm as written in Figure 1 is not self-stabilizing for mutual exclusion or deadlock freedom. It is easy to see that it is not self-stabilizing for deadlock freedom, since we could start all the processes in the **while** statement of the first **for** loop with their communication variables all *true*. It is not self-stabilizing

for mutual exclusion because a process could be started in its second **for** loop with its communication variable false, remain there arbitrarily long, waiting as higher numbered processes repeatedly execute their critical sections, and then execute its critical section while another process is also executing its critical section.

The One-Bit Algorithm is made self-stabilizing for both mutual exclusion and deadlock freedom by modifying each of the **while** loops so they read the value of x_i and correct its value if necessary. In other words, we place

if x_i **then** $x_i := \text{false}$

in the body of the first **for** loop's **while** statement, and likewise for the second **for** loop (except setting x_i true there). We now prove that this modification makes the One-Bit Algorithm self-stabilizing for mutual exclusion and deadlock freedom.

Theorem 4 With the above modification, the One-Bit Algorithm is self-stabilizing for mutual exclusion and deadlock freedom.

Proof It is easy to check that the proof of the deadlock freedom property in Theorem .3 is valid under the behavior assumed for self-stabilization, so the algorithm is self-stabilizing for deadlock freedom. To prove that it is self-stabilizing for mutual exclusion, we have to show that the mutual exclusion protocol is followed after a bounded number of system steps. It is easy to verify that this is true so long as every process that is in its second **for** loop (or past the point where it has decided to enter its second **for** loop) exits from that loop within a bounded number of system steps.³ We prove this by “backwards induction” on the process number.

To start the induction, we observe that since its second **for** loop is empty, process N must exit that **for** loop within some bounded number $t(N)$ of system steps. To complete the induction step, we assume that if $j > i$, then process j must exit its second **for** loop within $t(j)$ system steps of when it entered, and we prove that if process i is in its second **for** loop (after the *malfunction*), then it must eventually exit. We define the following sets:

S_1 : The set of processes waiting in their second **for** loop for a process numbered less than or equal to i .

S_2 : The set of processes waiting in their first **for** loop for a process in S_1 .

S_3 : The set of processes in their *trying* statement.

3. In this proof, we talk about where a process is in its program immediately before and after a system step. This makes sense because a system step contains an elementary operation from every process.

If process i is in its second **for** loop, then within a bounded number of steps it either leaves that loop or else sets x_i true. In the latter case, no process that then enters its *trying* section can leave it before process i does. Each higher-numbered process that is in its second **for** loop must leave it in a bounded number of system steps, whereupon any other process that is in its second **for** loop past the test of x_i must exit that loop within a bounded number of system steps. It follows that within a bounded number of steps, if process i is still in its second **for** loop, then the system execution reaches a point at which each of the three sets S_m cannot get smaller until i sets x_i false. It is then easy to see that once this has happened, within a bounded number of steps, one of the following must occur:

- Process i exits its second **for** loop.
- Another process joins the set S_3 .
- A process in S_3 joins S_1 or S_2 .

Since there are only N processes, there is a bound on how many times the second two can occur. Therefore, the first possibility must occur within a bounded number of system steps, completing the proof that process i must exit its second **for** loop within a bounded number of system steps. This in turn completes the proof of the theorem. ■

3.3 A Digression

Suppose N processes are arranged in a circle, with process 1 followed by process 2 followed by \dots followed by process N , which is followed by process 1. Each process communicates only with its two neighbors using an array v of boolean variables, each $v(i)$ being owned by process i and read by the following process. We want the processes to continue forever taking turns performing some action—first process 1, then process 2, and so on. Each process must be able to tell whether it is its turn by reading just its own variable $v(i)$ and that of the preceding process, and must pass the turn on to the next process by complementing the value of $v(i)$ (which is the only change it can make).

The basic idea is to let it be process i 's turn if the circle of variables $v(1), \dots, v(N)$ changes value at i —that is, if $v(i) = \neg v(i - 1)$. This doesn't quite work because a ring of values cannot change at only one point. However, we let process 1 be exceptional, letting it be 1's turn when $v(1) = v(N)$. The reader should convince himself that this works if all the $v(i)$ are initially equal.

It is obvious how this algorithm, which works for the cycle of all N processes arranged in order, is generalized to handle an arbitrary cycle of processes with one process singled out as the “first”. To describe the general algorithm more

formally, we need to introduce some notation. Recall that a *cycle* is an object of the form $\langle i_1, \dots, i_m \rangle$, where the i_j are distinct integers between 1 and N . The i_j are called the *elements* of the cycle. Two cycles are the same if they are identical except for a cyclic permutation of their elements—e.g., $\langle 1, 3, 5, 7 \rangle$ and $\langle 5, 7, 1, 3 \rangle$ are two representations of the same cycle, which is not the same cycle as $\langle 1, 5, 3, 7 \rangle$. We define the *first element* of a cycle to be its smallest element i_j .

By a *Boolean function on a cycle* we mean a Boolean function on its set of elements. The following functions are used in the remaining algorithms. Note that $CG(v, \gamma, i)$ is the Boolean function that has the value *true* if and only if it is process i 's turn to go next in the general algorithm applied to the cycle γ of processes.

Definition 2 Let v be a Boolean function on the cycle $\gamma = \langle i_1, \dots, i_m \rangle$, and let i_1 be the first element of γ . For each element i_j of the cycle we define:

$$CGV(v, \gamma, i_j) \stackrel{\text{def}}{=} \begin{cases} \neg v(i_{j-1}) & \text{if } j > 1; \\ v(i_m) & \text{if } j = 1. \end{cases}$$

$$CG(v, \gamma, i_j) \stackrel{\text{def}}{=} v(i_j) \equiv CGV(v, \gamma, i_j)$$

If $CG(v, \gamma, i_j)$ is true, then we say that v *changes value at i_j* .

The turn-taking algorithm, in which process i takes its turn when $CG(v, \gamma, i)$ equals *true*, works right when it is started with all the $v(i)$ having the same value. If it is started with arbitrary initial values for the $v(i)$, then several processes may be able to go at the same time. However, deadlock is impossible; it is always at least one process's turn. This is expressed formally by the following result, whose proof is simple and is left to the reader.

Lemma 3 Every Boolean function on a cycle changes value at some element—that is, for any Boolean function v on a cycle γ , there is some element i of γ such that $CG(v, \gamma, i) = \text{true}$.

We shall also need the following definitions. A cycle $\langle i_1, \dots, i_m \rangle$ is said to be *ordered* if, after a cyclic permutation of the i_j , $i_1 < \dots < i_m$. For example, $\langle 5, 7, 2, 3 \rangle$ is an ordered cycle while $\langle 2, 5, 3, 7 \rangle$ is not. Any nonempty set of integers in the range 1 to N defines a unique ordered cycle. If S is such a set, then we let $\text{ORD } S$ denote this ordered cycle.

3.4 The Three-Bit Algorithm

The One-Bit algorithm has the property that the lowest-numbered process that is trying to enter its critical section must eventually enter it—unless a still lower-numbered process enters the trying region before it can do so. However, a higher-numbered process can be locked out forever by lower-numbered processes repeatedly entering their critical sections. The basic idea of the Three-Bit Algorithm is for the processes' numbers to change in such a way that a waiting process must eventually become the lowest-numbered process.

Of course, we don't actually change a process's number. Rather, we modify the algorithm so that instead of process i 's two **for** loops running from 1 up to (but excluding) i and $i + 1$ to N , they run cyclically from f up to but excluding i and from $i \oplus 1$ up to but excluding f , where f is a function of the communication variables, and \oplus denotes addition modulo N . As processes pass through their critical sections, they change the value of their communication variables in such a way as to make sure that f eventually equals the number of every waiting process.

The first problem that we face in doing this is that if we simply replaced the **for** loops as indicated above, a process could be waiting inside its first **for** loop without ever discovering that f should have been changed. Therefore, we modify the algorithm so that when it finds x_j true, instead of waiting for it to become false, process i recomputes f and restarts its cycle of examining all the processes' communication variables.

We add two new communication variables to each process i . The variable y_i is set true by process i immediately upon entering its trying section, and is not set false until after process i has left its critical section and set x_i false. The variable z_i is complemented when process i leaves its critical section.

Finally, it is necessary to guarantee that while process i is in its trying region, a “lower-numbered” process that enters after i cannot enter its critical section before i does. This is accomplished by having the “lower-numbered” process wait for y_i to become false instead of x_i . This will still insure mutual exclusion, since x_i is false whenever y_i is.

Putting these changes all together, we get the algorithm of Figure 2. The “**for** $j := \dots$ **cyclically to** \dots ” denotes an iteration starting with j equal to the lower bound, and incrementing j by 1 modulo N up to but excluding the upper bound. We let $*:=*$ denote an assignment operation that performs a write only if it will change the value of the variable—i.e., the right-hand side is evaluated, compared with the current value of the variable on the left-hand side, and an assignment performed only if they are unequal. The $*:=*$ operation is introduced because we have assumed nothing about the value obtained by a read that is concurrent with a write, even if the write does not change the value. For example, executing $v := true$

```

private variables:  $j, f$  with range  $1 \dots N$ ,
 $\gamma$  with range cycles on  $1 \dots N$ ;
communication variables:  $x_i, y_i$  initially false,  $z_i$ ;
repeat forever
    noncritical section;
     $y_i := true$ ;
    l1:  $x_i := true$ ;
    l2:  $\gamma := \text{ORD}\{i : y_i = true\}$ ;
     $f := \text{minimum } \{j \in \gamma : CG(z, \gamma, j) = true\}$ ;
    for  $j := f$  cyclically to  $i$ 
        do if  $y_j$  then  $x_i *:=* false$ ;
        goto l2
    fi
    od;
    if  $\neg x_i$  then goto l1 fi;
    for  $j := i \oplus 1$  cyclically to  $f$ 
        do if  $x_j$  then goto l2 fi od;
    critical section;
     $z_i := \neg z_i$ ;
     $x_i := false$ ;
     $y_i := false$ 
end repeat

```

Figure 2 The three-bit algorithm: Process i

when v has the value *true* can cause a concurrent read of v to obtain the value *false*. However, executing $v *:=* true$ has absolutely no effect if v has the value *true*.

We let z denote the function that assigns the value z_j to j —so evaluating it at j requires a read of the variable z_j . Thus, $CG(z, \{1, 3, 5\}, 3) = true$ if and only if $z_1 \neq z_3$. Note that i is always an element of the cycle γ computed by process i , so the cycle is nonempty and the argument of the minimum function is a nonempty set (by Lemma 3).

We now prove that this algorithm satisfies the desired properties. In this and subsequent proofs, we will reason informally about processes looping and things happening eventually. The reader can refer to the proof of Theorem 3 to see how these arguments can be made more formal.

Theorem 5 The Three-Bit Algorithm satisfies the mutual exclusion, deadlock freedom and lockout freedom properties, and is shutdown safe and fail-safe for them.

Proof To verify the mutual exclusion property, we need only check that the basic mutual exclusion protocol is observed. This is not immediately obvious, since process i tests either x_j or y_j before entering its critical section, depending upon the value of f . However, a little thought will show that processes i and j do indeed follow the protocol before entering their critical sections, process i reading either x_j or y_j , and process j reading either x_i or y_i . This is true for the behavior allowed under shutdown and failure safety, so the algorithm is shutdown safe and fail-safe for mutual exclusion.

To prove deadlock freedom, assume for the sake of contradiction that some process executes a nonterminating *trying* statement, and that no process performs an infinite number of critical section executions. Then eventually there must be some set of processes looping forever in their *trying* statements, and all other processes forever executing their noncritical sections with their x and y variables false. Moreover, all the “*trying*” processes will eventually obtain the same value for f . Ordering the process numbers cyclically starting with f , let i be the lowest-numbered *trying* process. It is easy to see that all *trying* processes other than i will eventually set their x variables false, and i will eventually enter its critical section, providing the necessary contradiction.

We now show that the algorithm is lockout free. There must be a time at which one of the following three conditions is true for every process:

- It will execute its critical section infinitely many times.
- It is and will remain forever in its *trying* statement.
- It is and will remain forever in its noncritical section.

Suppose that this time has been reached, and let $\beta = \langle j_1, \dots, j_p \rangle$ be the ordered cycle formed from the set of processes for which one of the first two conditions holds. Note that we are not assuming j_1 to be the first element (smallest j_i) of β . We prove by contradiction that no process can remain forever in its *trying* section.

Suppose j_1 remains forever in its *trying* section. If j_2 were to execute its critical section infinitely many times, then it would eventually enter its *trying* section with z_{j_2} equal to $\neg CGV(z, \beta, j_2)$. When process j_2 then executes its statement $l2$, the cycle γ it computes will include the element j_1 , and it will compute $CG(z, \gamma, j_2)$ to equal *false*. It is easy to see that the value of f that j_2 then computes will cause j_1 to lie in the index range of its first **for** loop, so it must wait forever for process j_1 to set y_{j_1} *false*.

We therefore see that if j_1 remains forever in its *trying* section, then j_2 must also remain forever in its *trying* section. Since this is true for any element j_1 in the cycle β (we did not assume j_1 to be the first element), a simple induction argument

shows that if any process remains forever in its trying section, then all the processes j_1, \dots, j_p must remain forever in their trying sections. But this means that the system is deadlocked, which we have shown to be impossible, giving the required contradiction.

The above argument remains valid under shutdown and failure behavior, so the algorithm is shutdown safe and fail-safe for lockout freedom. ■

As with the One-Bit Algorithm, we must modify the Three-Bit Algorithm in order to make it self-stabilizing. It is necessary to make sure that process i does not wait in its *trying* section with y_i false. We therefore need to add the statement $y_i *:=* true$ somewhere between the label $l2$ and the beginning of the **for** statement. It is not necessary to correct the value of x_i because that happens automatically, and the initial value of z_i does not matter. We then have the following result.

Theorem 6 The Three-Bit Algorithm, with the above modification, is self-stabilizing for the mutual exclusion, deadlock freedom and lockout freedom properties.

Proof Within a bounded number of system steps, each process will either have passed through point $l2$ of its program twice, or entered its noncritical section and reset its x and y variables. (Remember that for self-stabilization, we must assume that these variables are reset in the noncritical section if they have the value *true*.) After that has happened, the system will be in a state it could have reached starting at the beginning from a normal initial state. ■

3.5 FCFS Solutions

We now describe two FCFS solutions. Both of them combine a mutual exclusion algorithm ME that is deadlock free but not FCFS with an algorithm FC that does not provide a mutual exclusion but does guarantee FCFS entry to its “critical section”. The mutual exclusion algorithm is embedded within the FCFS algorithm as follows.

```

repeat forever
    noncritical section;
    FC trying;
    FC critical section: begin
        ME trying;
        ME critical section;
        ME exit
    end;
    FC exit
end repeat

```

communication variables:

y_i initially false,
array z_i indexed by $\{1 \dots N\} - \{i\}$;

private variables:

array $after$ indexed by $\{1 \dots N\} - \{i\}$ of boolean,
 j with range $1 \dots N$;

repeat forever

noncritical section;

doorway: **for all** $j \neq i$

do $z_i[j] := \neg CGV(z_{ij}, \text{ORD}\{i, j\}, i)$ **od**;

for all $j \neq i$

do $after[j] := y_j$ **od**;

$y_i := \text{true}$;

waiting: **for all** $j \neq i$

do while $after[j]$

do if $CG(z_{ij}, \text{ORD}\{i, j\}, i) \vee \neg y_j$

then $after[j] := \text{false}$ **fi od**

od;

critical section;

$y_i := \text{false}$

end repeat

Figure 3 The N -Bit FCFS algorithm: Process i .

It is obvious that the entire algorithm satisfies the FCFS and mutual exclusion properties, where its doorway is the FC algorithm's doorway. Moreover, if both FC and ME are deadlock free, then the entire algorithm is also deadlock free. This is also true under shutdown and failure. Hence, if FC is shutdown safe (or fail-safe) for FCFS and deadlock freedom, and ME is shutdown safe (fail-safe) for mutual exclusion and deadlock freedom, then the entire algorithm is shutdown safe (fail-safe) for FCFS, mutual exclusion and deadlock freedom.

We can let ME be the One-Bit Algorithm, so we need only look for algorithms that are FCFS and deadlock free. The first one is the N -Bit Algorithm of Figure 3, which is a modification of an algorithm due to Katseff [5]. It uses N communication variables for each process. However, each of the $N - 1$ variables $z_i[j]$ of process i is read only by process j . Hence, the complete mutual exclusion algorithm using it and the One-Bit Algorithm requires the same number of single-reader variables as the Three-Bit Algorithm. The “**for all** j ” statement executes its body once for

each of the indicated values of j , with the separate executions done in any order (or interleaved). The function z_{ij} on the cycle $\text{ORD}\{i, j\}$ is defined by:

$$\begin{aligned} z_{ij}(i) &\stackrel{\text{def}}{=} z_i[j], \\ z_{ij}(j) &\stackrel{\text{def}}{=} z_j[i]. \end{aligned}$$

We now prove the following properties of this algorithm.

Lemma 4 The N -Bit Algorithm satisfies the FCFS and Deadlock Freedom properties, and is shutdown safe, abortion safe and fail-safe for them.

Proof Informally, the FCFS property is satisfied because if process i finishes its doorway before process j enters its doorway, but i has not yet exited, then j must see y_i true and wait for i to reset y_i or change $z_i[j]$. This argument is formalized as follows.

Assume that $\text{doorway}_i^{[k]} \rightarrow \text{doorway}_j^{[m]}$. Let $Y_i^{[k']}$ denote the write operation of y_i performed during $\text{doorway}_i^{[k]}$, let $Z_i[j]^{[k']}$ be the last write of $z_i[j]$ performed before $Y_i^{[k']}$.

We suppose that $CS_j^{[m]}$ exists, but that $CS_i^{[k]} \not\rightarrow CS_j^{[m]}$, and derive a contradiction. Let R be any read of y_i performed by process j during $\text{trying}_j^{[m]}$. Since $\text{doorway}_i^{[k]} \rightarrow \text{doorway}_j^{[m]}$, we have $Y_i^{[k']} \rightarrow R$. Since $CS_i^{[k]} \not\rightarrow CS_j^{[m]}$, A4 implies that $Y_i^{[k'+1]} \not\rightarrow R$. It then follows from C2 that the read R must obtain the value $y_i^{[k']}$, which equals *true*. A similar argument shows that every read of $z_i[j]$ during $\text{trying}_j^{[m]}$ obtains the value $z_i[j]^{[k'']}$. It is then easy to see that process j sets $\text{after}[i]$ true in its doorway and can never set it false because it always reads the same value of $z_i[j]$ in its *waiting* statement as it did in its *doorway*. Hence, j can never exit from its waiting section, which is the required contradiction.

We next prove deadlock freedom. The only way deadlock could occur is for there to be a cycle $\langle i_1, \dots, i_m \rangle$ of processes, each one waiting for the preceding one—i.e., with each process $i_{j \oplus 1}$ having $\text{after}[i_j]$ true. We assume that this is the case and obtain a contradiction. Let Ry_j denote the read of $y_{i_{j \oplus 1}}$ and let Wy_j denote the write of y_{i_j} by process i_j in the last execution of its doorway. Since $Ry_j \rightarrow Wy_j$ and the relation \rightarrow is acyclic, by A2 and A4 there must be some j such that $Wy_j \not\rightarrow Ry_{j \oplus 1}$. By C3, this implies that $Ry_{j \oplus 1} \rightarrow Wy_j$.

Let Wy' be the write of y_{i_j} that immediately precedes Wy_j , and thus sets its value false. If Wy' did not exist (because Wy_j was the first write of y_{i_j}) or $Ry_{j \oplus 1} \not\rightarrow Wy'$, it would follow from C2 and C3 that $Ry_{j \oplus 1}$ obtains the value *false*. But this is impossible because process $i_{j \oplus 1}$ has set $\text{after}[i_j]$ true. Hence, there is such a Wy' and $Ry_{j \oplus 1} \rightarrow Wy'$.

Using this result and A4, it is easy to check that the last write of $z_{i_{j+1}}[i_j]$ (during the last execution of the doorway of process i_{j+1}) must have preceded the reading of it by process i_j during the last execution of its doorway. It follows from this that in the deadlock state, $CG(z_{i_{j+1}}[i_j], \text{ORD}\{i_j, i_{j+1}\}, i_{j+1})$ must be true, contradicting the assumption that i_{j+1} is waiting forever with $\text{after}[i_j]$ true. This completes the proof of deadlock freedom. ■

We leave it to the reader to verify that the above proofs remain valid under shutdown, abortion and failure behavior. The only nontrivial part of the proof is showing that the algorithm is abortion safe for deadlock freedom. This property follows from the observation that if no process enters its critical section, then eventually all the values of $z_i[j]$ will stabilize and no more writes to those variables will occur—even if there are infinitely many abortions. ■

Using this lemma and the preceding remarks about embedding a mutual exclusion algorithm inside a FCFS algorithm, we can prove the following result.

Theorem 7 Embedding the One-Bit Algorithm inside the N -Bit Algorithm yields an algorithm that satisfies the mutual exclusion, FCFS, deadlock freedom and lockout freedom properties, and is shutdown safe, abortion safe and fail-safe for these properties.

Proof As we remarked above, the proof of the mutual exclusion, FCFS and deadlock freedom properties is trivial. Lockout freedom follows from these by Theorem 1. The fact that it is shutdown safe and fail-safe for these properties follows from the fact that the One-Bit and N -Bit algorithms are. The only thing left to show is that the entire algorithm is abortion safe for these properties even though the One-Bit algorithm is not. The FCFS property for the outer N -Bit algorithm implies that once a process has aborted, it cannot enter the One-Bit algorithm's *trying* statement until all the processes that were waiting there have either exited from the critical section or aborted. Hence, so far as the inner One-Bit algorithm is concerned, abortion is the same as shutdown until there are no more waiting processes. The shutdown safety of the One-Bit Algorithm therefore implies the abortion safety for the entire algorithm. ■

The above algorithm satisfies all of our conditions except for self-stabilization. It is not self-stabilizing for deadlock freedom because it is possible to start the algorithm in a state with a cycle of processes each waiting for the next. (The fact that this cannot happen in normal operation is due to the precise order in which variables are read and written.) In our final algorithm, we modify the N -Bit Algorithm to eliminate this possibility.

In the N -Bit Algorithm, process i waits for process j so long as the function z_{ij} on the cycle $\text{ORD}\{i, j\}$ does not change value at i . Since a function must change value at some element of a cycle, this prevents i and j from waiting for each other. However, it does not prevent a cycle of waiting processes containing more than two elements. We therefore introduce a function z_γ for every cycle γ , and we require that i wait for j only if for every cycle γ in which j precedes i : z_γ does not change value at i . It is easy to see that for any state, there can be no cycle γ in which each process waits for the preceding one, since z_γ must change value at some element of γ .

This leads us to the $N!$ -Bit Algorithm of Figure 4. We use the notation that $\text{CYC}(i)$ denotes the set of all cycles containing i and at least one other element, and $\text{CYC}(j, i)$ denotes the set of all those cycles in which j precedes i . We let z_γ denote the function on the cycle γ that assigns the value $z_i[\gamma]$ to the element i .

Using the N -Bit FCFS Algorithm, we can construct the “ultimate” algorithm that satisfies every property we have ever wanted from a mutual exclusion solution, as stated by the following theorem. Unfortunately, as the reader has no doubt noticed, this solution requires approximately $N!$ communication variables for each process, making it of little practical interest except for very small values of N .

Theorem 8 Embedding the One-Bit Algorithm inside the N -Bit Algorithm yields an algorithm that satisfies the mutual exclusion, FCFS, deadlock freedom and lockout freedom properties, and is shutdown safe, abortion safe, fail-safe and self-stabilizing for these properties.

Proof The proof of all but the self-stabilizing condition is the same as for the previous solution using the N -Bit Algorithm. It is easy to see that since the One-Bit Algorithm is self-stabilizing for mutual exclusion and deadlock freedom, to prove self-stabilization for the entire algorithm it suffices to prove that the N -Bit Algorithm is self-stabilizing for deadlock freedom. The proof of that is easily done using the above argument that there cannot be a cycle of processes each waiting endlessly for the preceding one. ■

4

Conclusion

Using the formalism of Part I, we stated the mutual exclusion problem, as well as several additional properties we might want a solution to satisfy. We then gave four algorithms, ranging from the inexpensive One-Bit Algorithm that satisfies only the most basic requirements to the ridiculously costly N -Bit Algorithm that satisfies every property we have ever wanted of a solution.

communication variables:

y_i initially false,
array z_i , indexed by $CYC(i)$;

private variables:

j with range $1 \dots N$;
 γ with range $CYC(i)$,
array $after$ indexed by $1 \dots N$ of booleans;

repeat forever

noncritical section;

doorway: **for all** $\gamma \in CYC(i)$ **do**
 $z_i[\gamma] := \neg CGV(z_\gamma, \gamma, j)$ **od**;

for all $j \neq i$
do $after[j] := y_j$ **od**;

waiting: **for all** $j \neq i$
do while $after[j]$
do $after[j] := y_j$;
for all $\gamma \in CYC(j, i)$
do if $\neg CG(z_\gamma, \gamma, i)$
then $after[j] := false$ **fi od**

od

od;

critical section;

$y_i := false$

end repeat

Figure 4 The $N!$ -Bit FCFS Algorithm: Process i .

Our proofs have been done in the style of standard “journal mathematics”, using informal reasoning that in principle can be reduced to very formal logic, but in practice never is. Our experience in years of devising synchronization algorithms has been that this style of proof is quite unreliable. We have on several occasions “proved” the correctness of synchronization algorithms only to discover later that they were incorrect. (Everyone working in this field seems to have the same experience.) This is especially true of algorithms using our nonatomic communication primitives.

This experience led us to develop a reliable method for proving properties of concurrent programs [9, 11, 16]. Instead of reasoning about a program’s behavior, one reasons in terms of its state. When the first version of the present paper

was written, it was not possible to apply this method to these mutual exclusion algorithms for the following reasons:

- The proof method required that the program be described in terms of atomic operations; we did not know how to reason about the nonatomic reads and writes used by the algorithms.
- Most of the correctness properties to be proved, as well as the properties assumed of the communication variables, were stated in terms of the program's behavior; we did not know how to apply our state-based reasoning to such behavioral properties.

Recent progress in reasoning about nonatomic operations [12] and in temporal logic specifications [13, 14] should make it possible to recast our definitions and proofs in this formalism. However, doing so would be a major undertaking, completely beyond the scope of this paper. We are therefore forced to leave these proofs in their current form as traditional, informal proofs. The behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable; we advise the reader to be skeptical of such proofs.

Acknowledgments. Many of these ideas have been maturing for quite a few years before appearing on paper for the first time here. They have been influenced by a number of people during that time, most notably Carel Scholten, Edsger Dijkstra, Chuck Seitz, Robert Keller, Irene Greif, and Michael Fischer. The impetus finally to write down the results came from discussions with Michael Rabin in 1980 that led to the discovery of the Three-Bit Algorithm.

References

- [1] BRINCH HANSEN, P. Concurrent programming concepts. *ACM Comput. Surv.* 5 (1973), 223–245.
- [1a] BURNS, J. Mutual exclusion with linear waiting using binary shared variables. *ACM SIGACT News* (Summer 1978), 42–47.
- [2] DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (Sept. 1965), 569.
- [3] DIJKSTRA, E. W. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11 (Nov. 1974), 643–644.
- [4] FISCHER, M. J., LYNCH, N., BURNS, J. E., AND BORODIN, A. Resource allocation with immunity to limited process failure. In *Proceedings of the 20th IEEE Symposium on the Foundations of Computer Science* (Oct.). IEEE, New York, 1979, pp. 234–254.

- [5] KATSEFF, H. P. A new solution to the critical section problem. In *Conference Record of the 10th Annual ACM Symposium on the Theory of Computing* (San Diego, Calif., May 1–3). ACM, New York, 1978, pp. 86–88.
- [6] KNUTH, D. E. Additional comments on a problem in concurrent program control. *Commun. ACM* 9, 5 (May 1966), 321–322.
- [7] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17, 8 (Aug. 1974), 453–455.
- [8] LAMPORT, L. The synchronization of independent processes. *Acta Inf.* 7, 1 (1976), 15–34.
- [9] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* SE-3, 2 (Mar. 1977), 125–143.
- [10] LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Netw.* 2 (1978), 95–114.
- [11] LAMPORT, L. The 'Hoare logic' of concurrent programs. *Acta Inf.* 14, 1 (1980), 21–37.
- [12] LAMPORT, L. Reasoning about nonatomic operations. In *Proceedings of the Tenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 24–26). ACM, New York, 1983, pp. 28–37.
- [13] LAMPORT, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.* 5, 2 (Apr. 1983), 190–222.
- [14] LAMPORT, L. What good is temporal logic? In *Information Processing 83: Proceedings of the IFIP 9th World Congress* (Paris, Sept. 19–23). R. E. A. Mason, Ed. North Holland, Amsterdam, 1983.
- [15] LAMPORT, L. The mutual exclusion problem: Part I—A theory of interprocess communication. *J. ACM* 33, 2 (Apr. 1986), 313–326.
- [16] OWICKI, S. and LAMPORT, L. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 455–495.
- [17] PETERSON, G. L. A new solution to Lamport's concurrent programming problem. *ACM Trans. Program. Lang. Syst.* 5, 1 (Jan. 1983), 56–65.
- [18] PETERSON, G. AND FISCHER, M. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing* (Boulder, Colo., May 2–4). ACM New York, 1977, pp. 91–97.
- [19] RABIN, M. The choice coordination problem. *Acta Inf.* 17 (1982), 121–134.
- [20] RIVEST, R. L. AND PRATT, V. R. The mutual exclusion problem for unreliable processes: Preliminary report. *Proceedings of the IEEE Symposium on the Foundation of Computer Science*. IEEE, New York, 1976, pp. 1–8.

RECEIVED DECEMBER 1980; REVISED SEPTEMBER 1985; ACCEPTED SEPTEMBER 1985

The Part-Time Parliament

Leslie Lamport (Digital Equipment Corporation)

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

Categories and Subject Descriptors: C.2.4 [Computer-Communications Networks]: Distributed Systems—*Network operating systems*; D.4.5 [Operating Systems]: Reliability—*Fault-tolerance*; J.1 [Computer Applications]: Administrative Data Processing—*Government*

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

Author's address: Systems Research, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

©1998 ACM 0734-2071/98/0500-0133 \$5.00

Paper originally published in *ACM Transactions on Computer Systems*, 16(2), May 1998, pp. 133–169.

1 The Problem

1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.¹ Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing. The Paxons' solution may therefore be of some interest to computer scientists. I present here a short history of the Paxos Parliament's protocol, followed by an even shorter discussion of its relevance for distributed systems.

Paxon civilization was destroyed by a foreign invasion, and archeologists have just recently begun to unearth its history. Our knowledge of the Paxon Parliament is therefore fragmentary. Although the basic protocols are known, we are ignorant of many details. Where such details are of interest, I will take the liberty of speculating on what the Paxons might have done.

1.2 Requirements

Parliament's primary task was to determine the law of the land, which was defined by the sequence of decrees it passed. A modern parliament will employ a secretary to record its actions, but no one in Paxos was willing to remain in the Chamber throughout the session to act as secretary. Instead, each Paxon legislator maintained a *ledger* in which he recorded the numbered sequence of decrees that were passed. For example, legislator Λινχό's ledger had the entry

155: *The olive tax is 3 drachmas per ton*

if she believed that the 155th decree passed by Parliament set the tax on olives to 3 drachmas per ton. Ledgers were written with indelible ink, and their entries could not be changed.

1. It should not be confused with the Ionian island of Paxoi, whose name is sometimes corrupted to *Paxos*.

Sidebar 1

This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

Keith Marzullo
University of California, San Diego

The first requirement of the parliamentary protocol was the *consistency of ledgers*, meaning that no two ledgers could contain contradictory information. If legislator $\Phi\iota\sigma\partial\epsilon\rho$ had the entry

132: *Lamps must use only olive oil*

in his ledger, then no other legislator's ledger could have a different entry for decree 132. However, another legislator might have no entry in his ledger for decree 132 if he hadn't yet learned that the decree had been passed.

Consistency of ledgers was not sufficient, since it could be trivially fulfilled by leaving all ledgers blank. Some requirement was needed to guarantee that decrees were eventually passed and recorded in ledgers. In modern parliaments, the passing of decrees is hindered by disagreement among legislators. This was not the case in Paxos, where an atmosphere of mutual trust prevailed. Paxon legislators were willing to pass any decree that was proposed. However, their peripatetic propensity posed a problem. Consistency would be lost if one group of legislators passed the decree

37: *Painting on temple walls is forbidden*

and then left for a banquet, whereupon a different group of legislators entered the Chamber and, knowing nothing about what had just happened, passed the conflicting decree

37: Freedom of artistic expression is guaranteed

Progress could not be guaranteed unless enough legislators stayed in the Chamber for a long enough time. Because Paxon legislators were unwilling to curtail their outside activities, it was impossible to ensure that any decree would ever be passed. However, legislators were willing to guarantee that, while in the Chamber, they and their aides would act promptly on all parliamentary matters. This guarantee allowed the Paxons to devise a parliamentary protocol satisfying the following *progress condition*.

If a majority of the legislators² were in the Chamber, and no one entered or left the Chamber for a sufficiently long period of time, then any decree proposed by a legislator in the Chamber would be passed, and every decree that had been passed would appear in the ledger of every legislator in the Chamber.

1.3 Assumptions

The requirements of the parliamentary protocol could be achieved only by providing the legislators with the necessary resources. Each legislator received a sturdy ledger in which to record the decrees, a pen, and a supply of indelible ink. Legislators might forget what they had been doing if they left the Chamber,³ so they would write notes in the back of the ledgers to remind themselves of important parliamentary tasks. An entry in the list of decrees was never changed, but notes could be crossed out. Achieving the progress condition required that legislators be able to measure the passage of time, so they were given simple hourglass timers.

Legislators carried their ledgers at all times, and could always read the list of decrees and any note that had not been crossed out. The ledgers were made of the finest parchment and were used for only the most important notes. A legislator would write other notes on a slip of paper, which he might (or might not) lose if he left the Chamber.

2. In translating the progress condition, I have rendered the Paxon word *μαδζδωριτσετ* as *majority of the legislators*. Alternative translations of this word have been proposed and are discussed in Section 2.2.

3. In one tragic incident, legislator *Tωνεγ* developed irreversible amnesia after being hit on the head by a falling statue just outside the Chamber.

The acoustics of the Chamber were poor, making oratory impossible. Legislators could communicate only by messenger, and were provided with funds to hire as many messengers as they needed. A messenger could be counted on not to garble messages, but he might forget that he had already delivered a message, and deliver it again. Like the legislators they served, messengers devoted only part of their time to parliamentary duties. A messenger might leave the Chamber to conduct some business—perhaps taking a six-month voyage—before delivering a message. He might even leave forever, in which case the message would never be delivered.

Although legislators and messengers could enter and leave at any time, when inside the Chamber they devoted themselves to the business of Parliament. While they remained in the Chamber, messengers delivered messages in a timely fashion, and legislators reacted promptly to any messages they received.

The official records of Paxos claim that legislators and messengers were scrupulously honest and strictly obeyed parliamentary protocol. Most scholars discount this as propaganda, intended to portray Paxos as morally superior to its eastern neighbors. Dishonesty, although rare, undoubtedly did occur. However, because it was never mentioned in official documents, we have little knowledge of how Parliament coped with dishonest legislators or messengers. What evidence has been uncovered is discussed in Section 3.3.5.

2 The Single-Decree Synod

The Paxon Parliament evolved from an earlier ceremonial Synod of priests that was convened every 19 years to choose a single, symbolic decree. For centuries, the Synod had chosen the decree by a conventional procedure that required all priests to be present. But as commerce flourished, priests began wandering in and out of the Chamber while the Synod was in progress. Finally, the old protocol failed, and a Synod ended with no decree chosen. To prevent a repetition of this theological disaster, Paxon religious leaders asked mathematicians to formulate a protocol for choosing the Synod's decree. The protocol's requirements and assumptions were essentially the same as those of the later Parliament except that instead of containing a sequence of decrees, a ledger would have at most one decree. The resulting Synod protocol is described here; the parliamentary protocol is described in Section 3.

Mathematicians derived the Synod protocol in a series of steps. First, they proved results showing that a protocol satisfying certain constraints would guarantee consistency and allow progress. A *preliminary protocol* was then derived directly from

these constraints. A restricted version of the preliminary protocol provided the *basic protocol* that guaranteed consistency, but not progress. The complete Synod protocol, satisfying the consistency and progress requirements, was obtained by restricting the basic protocol.⁴

The mathematical results are described in Section 2.1, and the protocols are described informally in Sections 2.2–2.4. A more formal description and correctness proof of the basic protocol appears in the appendix.

2.1 Mathematical Results

The Synod's decree was chosen through a series of numbered *ballots*, where a ballot was a referendum on a single decree. In each ballot, a priest had the choice only of voting for the decree or not voting.⁵ Associated with a ballot was a set of priests called a *quorum*. A ballot succeeded if and only if every priest in the quorum voted for the decree. Formally, a ballot B consisted of the following four components. (Unless otherwise qualified, *set* is taken to mean *finite set*.⁶)

- B_{dec} A decree (the one being voted on).
- B_{pot} A nonempty set of priests (the ballot's quorum).
- B_{vot} A set of priests (the ones who cast votes for the decree).⁷
- B_{bal} A ballot number.

A ballot B was said to be *successful* iff (if and only if) $B_{qrm} \subseteq B_{vot}$, so a successful ballot was one in which every quorum member voted.

Ballot numbers were chosen from an unbounded ordered set of numbers. If $B'_{bal} > B_{bal}$, then ballot B' was said to be *later* than ballot B . However, this indicated

4. The complete history of the Synod protocol's discovery is not known. Like modern computer scientists, Paxos mathematicians would describe elegant, logical derivations that bore no resemblance to how the algorithms were actually derived. However, it is known that the mathematical results (Theorems 1 and 2 of Section 2.1) really did precede the protocol. They were discovered when mathematicians, in response to the request for a protocol, were attempting to prove that a satisfactory protocol was impossible.

5. Like some modern nations, Paxos had not fully grasped the nature of Athenian democracy.

6. Although Paxos mathematicians were remarkably advanced for their time, they obviously had no knowledge of set theory. I have taken the liberty of translating the Paxos' more primitive notation into the language of modern set theory.

7. Only priests in the quorum actually voted, but Paxos mathematicians found it easier to convince people that the protocol was correct if, in their proof, they allowed any priest to vote in any ballot.

nothing about the order in which ballots were conducted; a later ballot could actually have taken place before an earlier one.

Paxon mathematicians defined three conditions on a set \mathcal{B} of ballots, and then showed that consistency was guaranteed and progress was possible if the set of ballots that had taken place satisfied those conditions. The first two conditions were simple; they can be stated informally as follows.

B1(\mathcal{B}) Each ballot in \mathcal{B} has a unique ballot number.

B2(\mathcal{B}) The quorums of any two ballots in \mathcal{B} have at least one priest in common.

The third condition was more complicated. One Paxon manuscript contained the following, rather confusing, statement of it.

B3(\mathcal{B}) For every ballot B in \mathcal{B} , if any priest in B 's quorum voted in an earlier ballot in \mathcal{B} , then the decree of B equals the decree of the latest of those earlier ballots.

Interpretation of this cryptic text was aided by the manuscript pictured in Figure 1, which illustrates condition *B3(\mathcal{B})* with a set \mathcal{B} of five ballots for a Synod consisting of the five priests A, B, Γ , Δ , and E. This set \mathcal{B} contains five ballots, where for each ballot the set of voters is the subset of the priests in the quorum whose names are enclosed in boxes. For example, ballot number 14 has decree α , a quorum containing three priests, and a set of two voters. Condition *B3(\mathcal{B})* has the form

#	decree	quorum and voters			
2	α	A	B	Γ	Δ
5	β	A	B	Γ	E
14	α	B		Δ	E
27	β	A		Γ	Δ
29	β		B	Γ	Δ

Figure 1 Paxon manuscript showing a set \mathcal{B} , consisting of five ballots, that satisfies conditions *B1(\mathcal{B})*–*B3(\mathcal{B})*. (Explanatory column headings have been added.)

“for every B in $\mathcal{B} : \dots$ ”, where “ \dots ” is a condition on ballot B . The conditions for the five ballots B of Figure 1 are as follows.

2. Ballot number 2 is the earliest ballot, so the condition on that ballot is trivially true.
5. None of ballot 5’s four quorum members voted in an earlier ballot, so the condition on ballot 5 is also trivially true.
14. The only member of ballot 14’s quorum to vote in an earlier ballot is Δ , who voted in ballot number 2, so the condition requires that ballot 14’s decree must equal ballot 2’s decree.
27. (This is a successful ballot.) The members of ballot 27’s quorum are A , Γ , and Δ . Priest A did not vote in an earlier ballot, the only earlier ballot Γ voted in was ballot 5, and the only earlier ballot Δ voted in was ballot 2. The latest of these two earlier ballots is ballot 5, so the condition requires that ballot 27’s decree must equal ballot 5’s decree.
29. The members of ballot 29’s quorum are B , Γ , and Δ . The only earlier ballot that B voted in was number 14, priest Γ voted in ballots 5 and 27, and Δ voted in ballots 2 and 27. The latest of these four earlier ballots is number 27, so the condition requires that ballot 29’s decree must equal ballot 27’s decree.

To state $B1(\mathcal{B})$ – $B3(\mathcal{B})$ formally requires some more notation. A *vote* v was defined to be a quantity consisting of three components: a priest v_{pst} , a ballot number v_{bal} , and a decree v_{dec} . It represents a vote cast by priest v_{pst} for decree v_{dec} in ballot number v_{bal} . The Paxons also defined *null* votes to be votes v with $v_{bal} = -\infty$ and $v_{dec} = \text{BLANK}$, where $-\infty < b < \infty$ for any ballot number b , and BLANK is not a decree. For any priest p , they defined null_p to be the unique null vote v with $v_{pst} = p$.

Paxon mathematicians defined a total ordering on the set of all votes, but part of the manuscript containing the definition has been lost. The remaining fragment indicates that, for any votes v and v' , if $v_{bal} < v'_{bal}$ then $v < v'$. It is not known how the relative order of v and v' was defined if $v_{bal} = v'_{bal}$.

For any set \mathcal{B} of ballots, the set $\text{Votes}(\mathcal{B})$ of votes in \mathcal{B} was defined to consist of all votes v such that $v_{pst} \in B_{pst}$, $v_{bal} \in B_{bal}$, and $v_{dec} \in B_{dec}$ for some $B \in \mathcal{B}$. If p is a priest and b is either a ballot number or $\pm\infty$, then $\text{MaxVote}(b, p, \mathcal{B})$ was defined to be the largest vote v in $\text{Votes}(\mathcal{B})$ cast by p with $v_{bal} < b$, or to be null_p if there was no such vote. Since null_p is smaller than any real vote cast by p , this means that $\text{MaxVote}(b, p, \mathcal{B})$ is the largest vote in the set

$$\{v \in \text{Votes}(\mathcal{B}) : (v_{pst} = p) \wedge (v_{bal} < b)\} \cup \{\text{null}_p\}.$$

For any nonempty set Q of priests, $\text{MaxVote}(b, Q, \mathcal{B})$ was defined to equal the maximum of all votes $\text{MaxVote}(b, p, \mathcal{B})$ with p in Q .

Conditions $B1(\mathcal{B})$ – $B3(\mathcal{B})$ are stated formally as follows.⁸

$$B1(\mathcal{B}) \triangleq \forall B, B' \in \mathcal{B} : (B \neq B') \Rightarrow (B_{bal} \neq B'_{bal})$$

$$B2(\mathcal{B}) \triangleq \forall B, B' \in \mathcal{B} : B_{qrm} \cap B'_{qrm} \neq \emptyset$$

$$B3(\mathcal{B}) \triangleq \forall B \in \mathcal{B} : (\text{MaxVote}(B_{bal}, B_{qrm}, \mathcal{B}_{bal}) \neq -\infty) \Rightarrow$$

$$B_{dec} = \text{MaxVote}(B_{bal}, B_{qrm}, \mathcal{B}_{dec})$$

Although the definition of MaxVote depends upon the ordering of votes, $B1(\mathcal{B})$ implies that $\text{MaxVote}(b, Q, \mathcal{B})_{dec}$ is independent of how votes with equal ballot numbers were ordered.

To show that these conditions imply consistency, the Paxons first showed that $B1(\mathcal{B})$ – $B3(\mathcal{B})$ imply that, if a ballot B in \mathcal{B} is successful, then any later ballot in \mathcal{B} is for the same decree as B .

Lemma If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{bal} > B_{bal})) \Rightarrow (B'_{dec} = B_{dec})$$

for any B, B' in \mathcal{B} .

Proof of Lemma For any ballot B in \mathcal{B} , let $\Psi(B, \mathcal{B})$ be the set of ballots in \mathcal{B} later than B for a decree different from B 's:

$$\Psi(B, \mathcal{B}) \triangleq \{B' \in \mathcal{B} : (B'_{bal} > B_{bal}) \wedge (B'_{dec} \neq B_{dec})\}$$

To prove the lemma, it suffices to show that if $B_{qrm} \subseteq B_{vot}$ then $\Psi(B, \mathcal{B})$ is empty. The Paxons gave a proof by contradiction. They assumed the existence of a B with $B_{qrm} \subseteq B_{vot}$ and $\Psi(B, \mathcal{B}) \neq \emptyset$, and obtained a contradiction as follows.⁹

1. Choose $C \in \Psi(B, \mathcal{B})$ such that $C_{bal} = \min\{B'_{bal} : B' \in \Psi(B, \mathcal{B})\}$.

PROOF: C exists because $\Psi(B, \mathcal{B})$ is nonempty and finite.

2. $C_{bal} > B_{bal}$

PROOF: By (1) and the definition of $\Psi(B, \mathcal{B})$.

3. $B_{vot} \cap C_{qrm} \neq \emptyset$

PROOF: By $B2(\mathcal{B})$ and the hypothesis that $B_{qrm} \subseteq B_{vot}$.

8. I use the Paxon mathematical symbol \triangleq , which means *equals by definition*.

9. Paxon mathematicians always provided careful, structured proofs of important theorems. They were not as sophisticated as modern mathematicians, who can omit many details and write paragraph-style proofs without ever making a mistake.

4. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \geq B_{bal}$
PROOF: By (2), (3), and the definition of $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})$.
5. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B}) \in \text{Votes}(\mathcal{B})$
PROOF: By (4) (which implies that $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})$ is not a null vote) and the definition of $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})$.
6. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{dec} = C_{dec}$
PROOF: By (5) and $B3(\mathcal{B})$.
7. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{dec} \neq B_{dec}$
PROOF: By (6), (1), and the definition of $\Psi(\mathcal{B}, \mathcal{B})$.
8. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{bal} > B_{bal}$
PROOF: By (4), since (7) and $B1(\mathcal{B})$ imply that $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{bal} \neq B_{bal}$.
9. $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B}) \in \text{Votes}(\Psi(\mathcal{B}, \mathcal{B}))$
PROOF: By (7), (8), and the definition of $\Psi(\mathcal{B}, \mathcal{B})$.
10. (10) $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})_{bal} < C_{bal}$
PROOF: By definition of $\text{MaxVote}(C_{bal}, C_{qrm}, \mathcal{B})$.
11. Contradiction
PROOF: By (9), (10), and (1). ■

With this lemma, it was easy to show that, if $B1-B3$ hold, then any two successful ballots are for the same decree.

Theorem 1 If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then

$$((B_{qrm} \subseteq B_{vot}) \wedge (B'_{qrm} \subseteq B'_{vot})) \Rightarrow (B'_{dec} = B_{dec})$$

for any B, B' in \mathcal{B} .

Proof of Theorem If $B'_{bal} = B_{bal}$, then $B1(\mathcal{B})$ implies $B' = B$. If $B'_{bal} \neq B_{bal}$, then the theorem follows immediately from the lemma. ■

The Paxons then proved a theorem asserting that if there are enough priests in the Chamber, then it is possible to conduct a successful ballot while preserving $B1-B3$. Although this does not guarantee progress, it at least shows that a balloting protocol based on $B1-B3$ will not deadlock.

Theorem 2 Let b be a ballot number, and let Q be a set of priests such that $b > B_{bal}$ and $Q \cap B_{qrm} \neq \emptyset$ for all $B \in \mathcal{B}$. If $B1(\mathcal{B})$, $B2(\mathcal{B})$, and $B3(\mathcal{B})$ hold, then there is a ballot

B' with $B'_{bal} = b$ and $B'_{qrm} = B'_{vot} = Q$ such that $B1(\mathcal{B} \cup \{B'\})$, $B2(\mathcal{B} \cup \{B'\})$, and $B3(\mathcal{B} \cup \{B'\})$ hold.

Proof of Theorem

Condition $B1(\mathcal{B} \cup \{B'\})$ follows from $B1(\mathcal{B})$, the choice of B'_{bal} , and the assumption about b . Condition $B2(\mathcal{B} \cup \{B'\})$ follows from $B2(\mathcal{B})$, the choice of B'_{qrm} , and the assumption about Q . If $MaxVote(b, Q, \mathcal{B})_{bal} = -\infty$ then let B'_{dec} be any decree. Else let it equal $MaxVote(b, Q, \mathcal{B})_{dec}$. Condition $B3(\mathcal{B} \cup \{B'\})$ then follows from $B3(\mathcal{B})$. ■

2.2 The Preliminary Protocol

The Paxons derived the *preliminary protocol* from the requirement that conditions $B1(\mathcal{B})$ – $B3(\mathcal{B})$ remain true, where \mathcal{B} was the set of all ballots that had been or were being conducted. The definition of the protocol specified how the set \mathcal{B} changed, but the set was never explicitly calculated. The Paxons referred to \mathcal{B} as a quantity observed only by the gods, since it might never be known to any mortal.

Each ballot was initiated by a priest, who chose its number, decree, and quorum. Each priest in the quorum then decided whether or not to vote in the ballot. The rules determining how the initiator chose a ballot's number, decree, and quorum, and how a priest decided whether or not to vote in a ballot were derived directly from the need to maintain $B1(\mathcal{B})$ – $B3(\mathcal{B})$.

To maintain $B1$, each ballot had to receive a unique number. By remembering (with notes in his ledger) what ballots he had previously initiated, a priest could easily avoid initiating two different ballots with the same number. To keep different priests from initiating ballots with the same number, the set of possible ballot numbers was partitioned among the priests. While it is not known how this was done, an obvious method would have been to let a ballot number be a pair consisting of an integer and a priest, using a lexicographical ordering, where

$$(13, \Gamma\rho\alpha\check{\iota}) < (13, \Lambda\iota\nu\sigma\epsilon\check{\iota}) < (15, \Gamma\rho\alpha\check{\iota})$$

since Γ came before Λ in the Paxon alphabet. In any case, it is known that every priest had an unbounded set of ballot numbers reserved for his use.

To maintain $B2$, a ballot's quorum was chosen to contain a $\mu\alpha\delta\zeta\partial\omega\rho\iota\tau\iota\sigma\epsilon\tau$ of priests. Initially, $\mu\alpha\delta\zeta\partial\omega\rho\iota\tau\iota\sigma\epsilon\tau$ just meant a simple majority. Later, it was observed that fat priests were less mobile and spent more time in the Chamber than thin ones, so a $\mu\alpha\delta\zeta\partial\omega\rho\iota\tau\iota\sigma\epsilon\tau$ was taken to mean any set of priests whose total weight was more than half the total weight of all priests, rather than a simple majority of the priests. When a group of thin priests complained that this was unfair, actual weights were replaced with symbolic weights based on a priest's

attendance record. The primary requirement for a $\mu\alpha\delta\zeta\partial\omega\rho\iota\tau\iota\sigma\epsilon\tau$ was that any two sets containing a $\mu\alpha\delta\zeta\partial\omega\rho\iota\tau\iota\sigma\epsilon\tau$ of priests had at least one priest in common. To maintain $B2$, the priest initiating a ballot B chose $B_{q_{rm}}$ to be a majority set.

Condition $B3$ requires that if $\text{MaxVote}(b, Q, \mathcal{B})_{dec}$ is not equal to BLANK, then a ballot with number b and quorum Q must have decree $\text{MaxVote}(b, Q, \mathcal{B})_{dec}$. If $\text{MaxVote}(b, Q, \mathcal{B})_{dec}$ equals BLANK, then the ballot can have any decree. To maintain $B3(\mathcal{B})$, before initiating a new ballot with ballot number b and quorum Q , a priest p had to find $\text{MaxVote}(b, Q, \mathcal{B})_{dec}$. To do this, p had to find $\text{MaxVote}(b, Q, \mathcal{B})$ for each priest q in Q .

Recall that $\text{MaxVote}(b, Q, \mathcal{B})$ is the vote with the largest ballot number less than b among all the votes cast by q , or $null_q$ if q did not vote in any ballot numbered less than b . Priest p obtains $\text{MaxVote}(b, q, \mathcal{B})$ from q by an exchange of messages. Therefore, the first two steps in the protocol for conducting a single ballot initiated by p are:¹⁰

1. Priest p chooses a new ballot number b and sends a $\text{NextBallot}(b)$ message to some set of priests.
2. A priest q responds to the receipt of a $\text{NextBallot}(b)$ message by sending a $\text{LastVote}(b, v)$ message to p , where v is the vote with the largest ballot number less than b that q has cast, or his null vote $null_q$ if q did not vote in any ballot numbered less than b .

Priest q must use notes in the back of his ledger to remember what votes he had previously cast.

When q sends the $\text{LastVote}(b, v)$ message, v equals $\text{MaxVote}(b, q, \mathcal{B})$. But the set \mathcal{B} of ballots changes as new ballots are initiated and votes are cast. Since priest p is going to use v as the value of $\text{MaxVote}(b, q, \mathcal{B})$ when choosing a decree, to keep $B3(\mathcal{B})$ true it is necessary that $\text{MaxVote}(b, q, \mathcal{B})$ not change after q has sent the $\text{LastVote}(b, v)$ message. To keep $\text{MaxVote}(b, q, \mathcal{B})$ from changing, q must cast no new votes with ballot numbers between v_{bal} and b . By sending the $\text{LastVote}(b, v)$ message, q is promising not to cast any such vote. (To keep this promise, q must record the necessary information in his ledger.)

The next two steps in the balloting protocol (begun in step (1) by priest p) are:

10. Priests p and q could be the same. For simplicity, the protocol is described with p sending messages to himself in this case. In reality, a priest could talk to himself without the use of messengers.

3. After receiving a $LastVote(b, v)$ message from every priest in some majority set Q , priest p initiates a new ballot with number b , quorum Q , and decree d , where d is chosen to satisfy B3. He then records the ballot in the back of his ledger and sends a $BeginBallot(b, d)$ message to every priest in Q .
4. Upon receipt of the $BeginBallot(b, d)$ message, priest q decides whether or not to cast his vote in ballot number b . (He may not cast the vote if doing so would violate a promise implied by a $LastVote(b', v')$ message he has sent for some other ballot.) If q decides to vote for ballot number b , then he sends a $Voted(b, q)$ message to p and records the vote in the back of his ledger.

The execution of step (3) is considered to add a ballot B to \mathcal{B} , where $B_{bal} = b$, $B_{qrm} = Q$, $B_{vot} = \emptyset$ (no one has yet voted in this ballot), and $B_{dec} = d$. In step (4), if priest q decides to vote in the ballot, then executing that step is considered to change the set \mathcal{B} of ballots by adding q to the set B_{vot} of voters in the ballot $B \in \mathcal{B}$.

A priest has the option not to vote in step (4), even if casting a vote would not violate any previous promise. In fact, all the steps in this protocol are optional. For example, a priest q can ignore a $NextBallot(b)$ message instead of executing step (2). Failure to take an action can prevent progress, but it cannot cause any inconsistency because it cannot make $B1(\mathcal{B}) - B3(\mathcal{B})$ false. Since the only effect not receiving a message can have is to prevent an action from happening, message loss also cannot cause inconsistency. Thus, the protocol guarantees consistency even if priests leave the chamber or messages are lost.

Receiving multiple copies of a message can cause an action to be repeated. Except in step (3), performing the action a second time has no effect. For example, sending several $Voted(b, q)$ messages in step (4) has the same effect as sending just one. The repetition of step (3) is prevented by using the entry made in the back of the ledger when it is executed. Thus, the consistency condition is maintained even if a messenger delivers the same message several times.

Steps (1)–(4) describe the complete protocol for initiating a ballot and voting on it. All that remains is to determine the results of the balloting and announce when a decree has been selected. Recall that a ballot is successful iff every priest in the quorum has voted. The decree of a successful ballot is the one chosen by the Synod. The rest of the protocol is:

5. If p has received a $Voted(b, q)$ message from every priest q in Q (the quorum for ballot number b), then he writes d (the decree of that ballot) in his ledger and sends a $Success(d)$ message to every priest.
6. Upon receiving a $Success(d)$ message, a priest enters decree d in his ledger.

Steps (1)–(6) describe how an individual ballot is conducted. The preliminary protocol allows any priest to initiate a new ballot at any time. Each step maintains $B1(\mathcal{B})$ – $B3(\mathcal{B})$, so the entire protocol also maintains these conditions. Since a priest enters a decree in his ledger only if it is the decree of a successful ballot, Theorem 1 implies that the priests' ledgers are consistent. The protocol does not address the question of progress.

In step (3), if the decree d is determined by condition $B3$, then it is possible that this decree is already written in the ledger of some priest. That priest need not be in the quorum Q ; he could have left the Chamber. Thus, consistency would not be guaranteed if step (3) allowed any greater freedom in choosing d .

2.3 The Basic Protocol

In the preliminary protocol, a priest must record (1) the number of every ballot he has initiated, (2) every vote he has cast, and (3) every *LastVote* message he has sent. Keeping track of all this information would have been difficult for the busy priests. The Paxons therefore restricted the preliminary protocol to obtain the more practical *basic protocol* in which each priest p had to maintain only the following information in the back of his ledger:

- $lastTried[p]$ The number of the last ballot that p tried to initiate, or $-\infty$ if there was none.
- $prevVote[p]$ The vote cast by p in the highest-numbered ballot in which he voted, or $-\infty$ if he never voted.
- $nextBal[p]$ The largest value of b for which p has sent a $lastVote(b, v)$ message, or $-\infty$ if he has never sent such a message.

Steps (1)–(6) of the preliminary protocol describe how a single ballot is conducted by its initiator, priest p . The preliminary protocol allows p to conduct any number of ballots concurrently. In the basic protocol, he conducts only one ballot at a time—ballot number $lastTried[p]$. After p initiates this ballot, he ignores messages that pertain to any other ballot that he had previously initiated. Priest p keeps all information about the progress of ballot number $lastTried[p]$ on a slip of paper. If he loses that slip of paper, then he stops conducting the ballot.

In the preliminary protocol, each $lastVote(b, v)$ message sent by a priest q represents a promise not to vote in any ballot numbered between v_{bal} and b . In the basic protocol, it represents the stronger promise not to cast a new vote in any ballot numbered less than b . This stronger promise might prevent him from casting a vote in step 4 of the basic protocol that he would have been allowed to cast in the preliminary protocol. However, since the preliminary protocol always gives

q the option of not casting his vote, the basic protocol does not require him to do anything not allowed by the preliminary protocol.

Steps (1)–(6) of the preliminary protocol become the following six steps for conducting a ballot in the basic protocol. (All information used by p to conduct the ballot, other than $lastTried[p]$, $prevVote[p]$, and $nextBal[p]$, is kept on a slip of paper.)

1. Priest p chooses a new ballot number b greater than $lastTried[p]$, sets $lastTried[p]$ to b , and sends a $NextBallot(b)$ message to some set of priests.
2. Upon receipt of a $NextBallot(b)$ message from p with $b > nextBal[q]$, priest q sets $nextBal[q]$ to b and sends a $LastVote(b, v)$ message to p , where v equals $prevVote[q]$. (A $NextBallot(b)$ message is ignored if $b \leq nextBal[q]$.)
3. After receiving a $LastVote(b, v)$ message from every priest in some majority set Q , where $b = lastTried[p]$, priest p initiates a new ballot with number b , quorum Q , and decree d , where d is chosen to satisfy $B3$. He then sends a $BeginBallot(b, d)$ message to every priest in Q .
4. Upon receipt of a $BeginBallot(b, d)$ message with $b = nextBal[q]$, priest q casts his vote in ballot number b , sets $prevVote[q]$ to this vote, and sends a $Voted(b, q)$ message to p . (A $BeginBallot(b, d)$ message is ignored if $b \neq nextBal[q]$.)
5. If p has received a $Voted(b, q)$ message from every priest q in Q (the quorum for ballot number b), where $b = lastTried[p]$, then he writes d (the decree of that ballot) in his ledger and sends a $Success(d)$ message to every priest.
6. Upon receiving a $Success(d)$ message, a priest enters decree d in his ledger.

The basic protocol is a restricted version of the preliminary protocol, meaning that every action allowed by the basic protocol is also allowed by the preliminary protocol. Since the preliminary protocol satisfies the consistency condition, the basic protocol also satisfies that condition. Like the preliminary protocol, the basic protocol does not require that any action ever be taken, so it does not address the question of progress.

The derivation of the basic protocol from $B1$ – $B3$ made it obvious that the consistency condition was satisfied. However, some similarly “obvious” ancient wisdom had turned out to be false, and skeptical citizens demanded a more rigorous proof. Their Paxon mathematicians’ proof that the protocol satisfies the consistency condition is reproduced in the appendix.

2.4 The Complete Synod Protocol

The basic protocol maintains consistency, but it cannot ensure any progress because it states only what a priest may do; it does not require him to do anything. The complete protocol consists of the same six steps for conducting a ballot as the basic protocol. To help achieve progress, it includes the obvious additional requirement that priests perform steps (2)–(6) of the protocol as soon as possible. However, to meet the progress condition, it is necessary that some priest be required to perform step (1), which initiates a ballot. The key to the complete protocol lay in determining when a priest should initiate a ballot.

Never initiating a ballot will certainly prevent progress. However, initiating too many ballots can also prevent progress. If b is larger than any other ballot number, then the receipt of a $NextBallot(b)$ message by priest q in step (2) may elicit a promise that prevents him from voting in step 4 for any previously initiated ballot. Thus, the initiation of a new ballot can prevent any previously initiated ballot from succeeding. If new ballots are continually initiated with increasing ballot numbers before the previous ballots have a chance to succeed, then no progress might be made.

Achieving the progress condition requires that new ballots be initiated until one succeeds, but that they not be initiated too frequently. To develop the complete protocol, the Paxons first had to know how long it took messengers to deliver messages and priests to respond. They determined that a messenger who did not leave the Chamber would always deliver a message within 4 minutes, and a priest who remained in the Chamber would always perform an action within 7 minutes of the event that caused the action.¹¹ Thus, if p and q were in the Chamber when some event caused p to send a message to q , and q responded with a reply to p , then p would receive that reply within 22 minutes if neither messenger left the Chamber. (Priest p would send the message within 7 minutes of the event, q would receive the message within 4 more minutes, he would respond within 7 minutes, and the reply would reach p within 4 more minutes.)

Suppose that only a single priest p was initiating ballots, and that he did so by sending a message to every priest in step (1) of the protocol. If p initiated a ballot when a majority set of priests was in the chamber, then he could expect to execute step (3) within 22 minutes of initiating the ballot, and to execute step (5) within

11. I am assuming a value of 30 seconds for the $\delta\zeta\partial\iota\phi\check{r}$, the Paxon unit of time. This value is within the range determined from studies of hourglass shards. The reaction time of priests was so long because they had to respond to every message within 7 minutes (14 $\delta\zeta\partial\iota\phi\check{r}$), even if a number of messages arrived simultaneously.

another 22 minutes. If he was unable to execute the steps by those times, then either some priest or messenger left the Chamber after p initiated the ballot, or a larger-numbered ballot had previously been initiated by another priest (before p became the only priest to initiate ballots). To handle the latter possibility, p had to learn about any ballot numbers greater than $lastTried[p]$ used by other priests. This could be done by extending the protocol to require that if a priest q received a *NextBallot*(b) or a *BeginBallot*(b, d) message from p with $b < nextBal[q]$, then he would send p a message containing $nextBal[q]$. Priest p would then initiate a new ballot with a larger ballot number.

Still assuming that p was the only priest initiating ballots, suppose that he were required to initiate a new ballot iff (1) he had not executed step (3) or step (5) within the previous 22 minutes, or (2) he learned that another priest had initiated a higher-numbered ballot. If the Chamber doors were locked with p and a majority set of priests inside, then a decree would be passed and recorded in the ledgers of all priests in the Chamber within 99 minutes. (It could take 22 minutes for p to start the next ballot, 22 more minutes to learn that another priest had initiated a larger-numbered ballot, then 55 minutes to complete steps (1)–(6) for a successful ballot.) Thus, the progress condition would be met if only a single priest, who did not leave the chamber, were initiating ballots.

The complete protocol therefore included a procedure for choosing a single priest, called the *president*, to initiate ballots. In most forms of government, choosing a president can be a difficult problem. However, the difficulty arises only because most governments require that there be exactly one president at any time. In the United States, for example, chaos would result after the 1988 election if some people thought Bush had been elected president while others thought that Dukakis had, since one of them might decide to sign a bill into law while the other decided to veto it. However, in the Paxon Synod, having multiple presidents could only impede progress; it could not cause inconsistency. For the complete protocol to satisfy the progress condition, the method for choosing the president needed only to satisfy the following *presidential selection requirement*:

If no one entered or left the Chamber, then after T minutes exactly one priest in the Chamber would consider himself to be the president.

If the presidential selection requirement were met, then the complete protocol would have the property that if a majority set of priests were in the chamber and no one entered or left the Chamber for $T + 99$ minutes, then at the end of that period every priest in the Chamber would have a decree written in his ledger.

The Paxons chose as president the priest whose name was last in alphabetical order among the names of all priests in the Chamber, though we don't know exactly how this was done. The presidential selection requirement would have been satisfied if a priest in the Chamber sent a message containing his name to every other priest at least once every $T - 11$ minutes, and a priest considered himself to be president iff he received no message from a "higher-named" priest for T minutes.

The complete Synod protocol was obtained from the basic protocol by requiring priests to perform steps (2)–(6) promptly, adding a method for choosing a president who initiated ballots, and requiring the president to initiate ballots at the appropriate times. Many details of the protocol are not known. I have described simple methods for selecting a president and for deciding when the president should initiate a new ballot, but they are undoubtedly not the ones used in Paxos. The rules I have given require the president to keep initiating ballots even after a decree has been chosen, thereby ensuring that priests who have just entered the Chamber learn about the chosen decree. There were obviously better ways to make sure priests learned about the decree after it had been chosen. Also, in the course of selecting a president, each priest probably sent his value of $lastTried[p]$ to the other priests, allowing the president to choose a large enough ballot number on his first try.

The Paxons realized that any protocol to achieve the progress condition must involve measuring the passage of time.¹² The protocols given above for selecting a president and initiating ballots are easily formulated as precise algorithms that set timers and perform actions when time-outs occur—assuming perfectly accurate timers. A closer analysis reveals that such protocols can be made to work with timers having a known bound on their accuracy. The skilled glass blowers of Paxos had no difficulty constructing suitable hourglass timers.

Given the sophistication of Paxos mathematicians, it is widely believed that they must have found an optimal algorithm to satisfy the presidential selection requirement. We can only hope that this algorithm will be discovered in future excavations on Paxos.

3 The Multidecree Parliament

When Parliament was established, a protocol to satisfy its consistency and progress requirements was derived from the Synod protocol. The derivation and properties

12. However, many centuries were to pass before a rigorous proof of this result was given [Fischer et al. 1985].

of the original parliamentary protocol are described in Sections 3.1 and 3.2. Section 3.3 discusses the further evolution of the protocol.

3.1 The Protocol

Instead of passing just one decree, the Paxon Parliament had to pass a series of numbered decrees. As in the Synod protocol, a president was elected. Anyone who wanted a decree passed would inform the president, who would assign a number to the decree and attempt to pass it. Logically, the parliamentary protocol used a separate instance of the complete Synod protocol for each decree number. However, a single president was selected for all these instances, and he performed the first two steps of the protocol just once.

The key to deriving the parliamentary protocol is the observation that, in the Synod protocol, the president does not choose the decree or the quorum until step 3. A newly elected president p can send to some set of legislators a single message that serves as the $NextBallot(b)$ message for all instances of the Synod protocol. (There are an infinite number of instances—one for each decree number.) A legislator q can reply with a single message that serves as the $LastVote$ messages for step (2) of all instances of the Synod protocol. This message contains only a finite amount of information, since q can have voted in only a finite number of instances.

When the new president has received a reply from every member of a majority set, he is ready to perform step (3) for every instance of the Synod protocol. For some finite number of instances (decree numbers), the choice of decree in step (3) will be determined by $B3$. The president immediately performs step 3 for each of those instances to try passing these decrees. Then, whenever he receives a request to pass a decree, he chooses the lowest-numbered decree that he is still free to choose, and he performs step (3) for that decree number (instance of the Synod protocol) to try to pass the decree.

The following modifications to this simple protocol lead to the actual Paxon Parliament's protocol.

- There is no reason to go through the Synod protocol for a decree number whose outcome is already known. Therefore, if a newly elected president p has all decrees with numbers less than or equal to n written in his ledger, then he sends a $NextBallot(b, n)$ message that serves as a $NextBallot(b)$ message in all instances of the Synod protocol for decree numbers larger than n . In his response to this message, legislator q informs p of all decrees numbered greater than n that already appear in q 's ledger (in addition to sending the

usual *LastVote* information for decrees not in his ledger), and he asks p to send him any decrees numbered n or less that are not in his ledger.

- Suppose decrees 125 and 126 are introduced late Friday afternoon, decree 126 is passed and is written in one or two ledgers, but before anything else happens, the legislators all go home for the weekend. Suppose also that the following Monday, $\Delta\phi\omega\rho\kappa$ is elected the new president and learns about decree 126, but she has no knowledge of decree 125 because the previous president and all legislators who had voted for it are still out of the Chamber. She will hold a ballot that passes decree 126, which leaves a gap in the ledgers. Assigning number 125 to a new decree would cause it to appear earlier in the ledger than decree 126, which had been passed the previous week. Passing decrees out of order in this way might cause confusion—for example, if the citizen who proposed the new decree did so because he knew decree 126 had already passed. Instead, $\Delta\phi\omega\rho\kappa$ would attempt to pass

125: The ides of February is national olive day

a traditional decree that made absolutely no difference to anyone in Paxos. In general, a new president would fill any gaps in his ledger by passing the “olive-day” decree.

The consistency and progress properties of the parliamentary protocol follow immediately from the corresponding properties of the Synod protocol from which it was derived. To our knowledge, the Paxons never bothered writing a precise description of the parliamentary protocol because it was so easily derived from the Synod protocol.

3.2 Properties of the Protocol

3.2.1 The Ordering of Decrees

Balloting could take place concurrently for many different decree numbers, with ballots initiated by different legislators—each thinking he was president when he initiated the ballot. We cannot say precisely in what order decrees would be passed, especially without knowing how a president was selected. However, there is one important property about the ordering of decrees that can be deduced.

A decree was said to be *proposed* when it was chosen by the president in step (3) of the corresponding instance of the Synod protocol. The decree was said to be *passed* when it was written for the first time in a ledger. Before a president could propose any new decrees, he had to learn from all the members of a majority set what decrees they had voted for. Any decree that had already been passed must

have been voted for by at least one legislator in the majority set. Therefore, the president must have learned about all previously passed decrees before initiating any new decree. The president would not fill a gap in the ledgers with an important decree—that is, with any decree other than the “olive-day” decree. He would also not propose decrees out of order. Therefore, the protocol satisfied the following *decree-ordering property*.

If decrees *A* and *B* are important and decree *A* was passed before decree *B* was proposed, then *A* has a lower decree number than *B*.

3.2.2 Behind Closed Doors

Although we don’t know the details involved in choosing a new president, we do know exactly how Parliament functioned when the president had been chosen and no one was entering or leaving the Chamber. Upon receiving a request to pass a decree—either directly from a citizen or relayed from another legislator—the president assigned the decree a number and passed it with the following exchange of messages. (The numbers refer to the corresponding steps in the Synod protocol.)

3. The president sent a *BeginBallot* message to each legislator in a quorum.
4. Each legislator in the quorum sent a *Voted* message to the president.
5. The president sent a *Success* message to every legislator.

This is a total of three message delays and about $3N$ messages, assuming a parliament of N legislators and a quorum of about $N/2$. Moreover, if Parliament was busy, the president would combine the *BeginBallot* message for one decree with the *Success* message for a previous one, for a total of only $2N$ messages per decree.

3.3 Further Developments

Governing the island turned out to be a more complex task than the Paxons realized. A number of problems arose whose solutions required changes to the protocol. The most important of these changes are described below.

3.3.1 Picking a President

The president of parliament was originally chosen by the method that had been used in the Synod, which was based purely on the alphabetical ordering of names. Thus, when legislator Ω_{K1} returned from a six-month vacation, he was immediately made president—even though he had no idea what had happened in his absence. Parliamentary activity came to a halt while Ω_{K1} , who was a slow writer, laboriously copied six months worth of decrees to bring his ledger up to date.

This incident led to a debate about the best way to choose a president. Some Paxons urged that once a legislator became president, he should remain president until he left the Chamber. An influential group of citizens wanted the richest legislator in the Chamber to be president, since he could afford to hire more scribes and other servants to help him with the presidential duties. They argued that once a rich legislator had brought his ledger up to date, there was no reason for him not to assume the presidency. Others, however, argued that the most upstanding citizen should be made president, regardless of wealth. Upstanding probably meant less likely to be dishonest, although no Paxon would publicly admit the possibility of official malfeasance. Unfortunately, the outcome of this debate is not known; no record exists of the presidential selection protocol that was ultimately used.

3.3.2 Long Ledgers

As the years progressed and Parliament passed more and more decrees, Paxons had to pore over an ever longer list of decrees to find the current olive tax or what color goat could be sold. A legislator who returned to the Chamber after an extended voyage had to do quite a bit of copying to bring his ledger up to date. Eventually, the legislators were forced to convert their ledgers from lists of decrees into law books that contained only the current state of the law and the number of the last decree whose passage was reflected in that state.

To learn the current olive tax, one looked in the law book under “taxes;” to learn what color goat could be sold, one looked under “mercantile law.” If a legislator’s ledger contained the law through decree 1298 and he learned that decree 1299 set the olive tax to 6 drachmas per ton, he just changed the entry for the olive-tax law and noted that his ledger was complete through decree 1299. If he then learned about decree 1302, he would write it down in the back of the ledger and wait until he learned about decrees 1300 and 1301 before incorporating decree 1302 into the law book.

To enable a legislator who had been gone for a short time to catch up without copying the entire law book, legislators kept a list of the past week’s decrees in the back of the book. They could have kept this list on a slip of paper, but it was convenient for a legislator to enter decrees in the back of the ledger as they were passed and update the law book only two or three times a week.

3.3.3 Bureaucrats

As Paxos prospered, legislators became very busy. Parliament could no longer handle all details of government, so a bureaucracy was established. Instead of passing

a decree to declare whether each lot of cheese was fit for sale, Parliament passed a decree appointing a cheese inspector to make those decisions.

It soon became evident that selecting bureaucrats was not as simple as it first seemed. Parliament passed a decree making $\Delta\kappa\sigma\tau\rho\alpha$ the first cheese inspector. After some months, merchants complained that $\Delta\kappa\sigma\tau\rho\alpha$ was too strict and was rejecting perfectly good cheese. Parliament then replaced him by passing the decree

1375: $\Gamma\omega\nu\delta\alpha$ is the new cheese inspector

But $\Delta\kappa\sigma\tau\rho\alpha$ did not pay close attention to what Parliament did, so he did not learn of this decree right away. There was a period of confusion in the cheese market when both $\Delta\kappa\sigma\tau\rho\alpha$ and $\Gamma\omega\nu\delta\alpha$ were inspecting cheese and making conflicting decisions.

To prevent such confusion, the Paxons had to guarantee that a position could be held by at most one bureaucrat at any time. To do this, a president included as part of each decree the time and date when it was proposed. A decree making $\Delta\kappa\sigma\tau\rho\alpha$ the cheese inspector might read

2716: 8:30 15 Jan 72— $\Delta\kappa\sigma\tau\rho\alpha$ is cheese inspector for 3 months

This declares his term to begin either at 8:30 on 15 January or when the previous inspector's term ended—whichever was later. His term would end at 8:30 on 15 March, unless he explicitly resigned by asking the president to pass a decree like

2834 9:15 3 Mar 72— $\Delta\kappa\sigma\tau\rho\alpha$ resigns as cheese inspector

A bureaucrat was appointed for a short term, so he could be replaced quickly—for example, if he left the island. Parliament would pass a decree to extend the bureaucrat's term if he was doing a satisfactory job.

A bureaucrat needed to tell time to determine if he currently held a post. Mechanical clocks were unknown on Paxos, but Paxons could tell time accurately to within 15 minutes by the position of the sun or the stars.¹³ If $\Delta\kappa\sigma\tau\rho\alpha$'s term began at 8:30, he would not start inspecting cheese until his celestial observations indicated that it was 8:45.

It is easy to make this method of appointing bureaucrats work if higher-numbered decrees always have later proposal times. But what if Parliament passed the decrees

2854: 9:45 9 Apr 78— $\Phi\rho\alpha\nu\sigma\epsilon\zeta$ is wine taster for 2 months

2855: 9:20 9 Apr 78— $\Pi\nu\nu\epsilon\lambda\iota$ is wine taster for 1 month

13. Cloudy days are rare in Paxos's balmy climate.

that were proposed between 9:30 and 9:35 by different legislators who both thought they were president? Such out-of-order proposal times are easily prevented because the parliamentary protocol satisfies the following property.

If two decrees are passed by different presidents, then one of the presidents proposed his decree after learning that the other decree had been proposed.

To see that this property is satisfied, suppose that ballot number b was successful for decree D , ballot number b' was successful for decree D' , and $b < b'$. Let q be a legislator who voted in both ballots. The balloting for D' began with a $NextBallot(b', n)$ message. If the sender of that message did not already know about D , then n is less than the decree number of D , and q 's reply to the $NextBallot$ message must state that he voted for D .

3.3.4 Learning the Law

In addition to requesting the passage of decrees, ordinary citizens needed to inquire about the current law of the land. The Paxons at first thought that a citizen could simply examine the ledger of any legislator, but the following incident demonstrated that a more sophisticated approach was needed. For centuries, it had been legal to sell only white goats. A farmer named $\Delta\omega\lambda\epsilon\phi$ got Parliament to pass the decree

77: The sale of black goats is permitted

$\Delta\omega\lambda\epsilon\phi$ then instructed his goatherd to sell some black goats to a merchant named $\Sigma\kappa\epsilon\epsilon\nu$. As a law-abiding citizen, $\Sigma\kappa\epsilon\epsilon\nu$ asked legislator $\Sigma\tau\omega\kappa\mu\epsilon\check{\iota}\rho$ if such a sale would be legal. But $\Sigma\tau\omega\kappa\mu\epsilon\check{\iota}\rho$ had been out of the Chamber and had no entry in his ledger past decree 76. He advised $\Sigma\kappa\epsilon\epsilon\nu$ that the sale would be illegal under the current law, so $\Sigma\kappa\epsilon\epsilon\nu$ refused to buy the goats.

This incident led to the formulation of the following *monotonicity condition* on inquiries about the law.

If one inquiry precedes a second inquiry, then the second inquiry cannot reveal an earlier state of the law than the first.

If a citizen learns that a particular decree has been passed, then the process of acquiring that knowledge is considered to be an implicit inquiry to which this condition applies. As we will see, the interpretation of the monotonicity condition changed over the years.

Initially, the monotonicity condition was achieved by passing a decree for each inquiry. If $\Sigma\delta\nu\check{\delta}\epsilon\rho$ wanted to know the current tax on olives, he would get Parliament to pass a decree such as

87: *Citizen $\Sigma\delta\nu\check{\delta}\epsilon\rho$ is reading the law*

He would then read any ledger complete at least through decree 86 to learn the olive tax as of that decree. If citizen $\Gamma\rho\epsilon\epsilon\varsigma$ then inquired about the olive tax, the decree for his inquiry was proposed after decree 87 was passed, so the decree-ordering property (Section 3.2.1) implies that it received a decree number greater than 87. Therefore, $\Gamma\rho\epsilon\epsilon\varsigma$ could not obtain an earlier value of the olive tax than $\Sigma\delta\nu\check{\delta}\epsilon\rho$. This method of reading the law satisfied the monotonicity condition when *precedes* was interpreted to mean that inquiry *A* precedes inquiry *B* iff *A* finished at an earlier time than *B* began.

Passing a decree for every inquiry soon proved too cumbersome. The Paxons realized that a simpler method was possible if they weakened the monotonicity condition by changing the interpretation of *precedes*. They decided that for one event to precede another, the first event not only had to happen at an earlier time, but it had to be able to causally affect the second event. The weaker monotonicity condition prevents the problem first encountered by farmer $\Delta\omega\lambda\epsilon\phi$ and merchant $\Sigma\kappa\epsilon\epsilon\nu$ because there is a causal chain of events between the end of the implicit inquiry by $\Delta\omega\lambda\epsilon\phi$ and the beginning of the inquiry by $\Sigma\kappa\epsilon\epsilon\nu$.

The weaker monotonicity condition was met by using decree numbers in all business transactions and inquiries. For example, farmer $\Delta\omega\lambda\epsilon\phi$, whose flock included many nonwhite goats, got Parliament to pass the decree

277: *The sale of brown goats is permitted*

When selling his brown goats to $\Sigma\kappa\epsilon\epsilon\nu$, he informed the merchant that the sale was legal as of decree number 277. $\Sigma\kappa\epsilon\epsilon\nu$ then asked legislator $\Sigma\tau\omega\kappa\mu\epsilon\check{\iota}\rho$ if the sale were legal under the law through at least decree 277. If $\Sigma\tau\omega\kappa\mu\epsilon\check{\iota}\rho$'s ledger was not complete through decree 277, he would either wait until it was or else tell $\Sigma\kappa\epsilon\epsilon\nu$ to ask someone else. If $\Sigma\tau\omega\kappa\mu\epsilon\check{\iota}\rho$'s ledger went through decree 298, then he would tell $\Sigma\kappa\epsilon\epsilon\nu$ that the sale was legal as of decree number 298. Merchant $\Sigma\kappa\epsilon\epsilon\nu$ would remember the number 298 for use in his next business transaction or inquiry about the law.

The Paxons had satisfied the monotonicity condition, but ordinary citizens disliked having to remember decree numbers. Again, the Paxons solved the problem by reinterpreting the monotonicity condition—this time, by changing the meaning of *state of the law*. They divided the law into separate areas, and a legislator was

chosen as specialist for each area. The current state of each area of the law was determined by that specialist's ledger. For example, suppose decree 1517 changed the tariff law and decree 1518 changed the tax law. The tax law would change first if the tax-law specialist learned of both decrees before the tariff-law specialist learned of either, yielding a state of the law that could not be obtained by enacting the decrees in numerical order.

To avoid conflicting definitions of the current state, the Paxons required that there be at most one specialist at a time for any area. This requirement was satisfied by using the same method to choose specialists that was used to choose bureaucrats (see Section 3.3.3). If each inquiry involved only a single area of the law, monotonicity was then achieved by directing the inquiry to that area's specialist, who answered it from his ledger. Since learning that a law had passed constituted the result of an implicit inquiry, the Paxons required that a decree change at most one area of the law, and that notification of the decree's passage could come only from the area's specialist.

Inquiries involving multiple areas were not hard to handle. When merchant $\Lambda\iota\sigma\kappa\omega\phi$ asked if the tariff on an imported golden fleece was higher than the sales tax on one purchased locally, the tax-law and tariff-law specialists had to cooperate to provide an answer. For example, the tax specialist could answer $\Lambda\iota\sigma\kappa\omega\phi$ by first asking the tariff specialist for the tariff on golden fleeces, so long as he made no changes to his ledger before receiving a reply.

This method proved satisfactory until it became necessary to make a sweeping change to several areas of the law at one time. The Paxons then realized that the necessary requirement for maintaining monotonicity was not that a decree affect only a single area, but that every area it affects have the same specialist. Parliament could change several areas of the law with a single decree by first appointing a single legislator to be the specialist for all those areas. Moreover, the same area could have multiple specialists, so long as that area of the law was not allowed to change. Just before income taxes were due, Parliament would appoint several tax-law specialists to handle the seasonal flood of inquiries about the tax law.

3.3.5 Dishonest Legislators and Honest Mistakes

Despite official assertions to the contrary, there must have been a few dishonest legislators in the history of Paxos. When caught, they were probably exiled. By sending contradictory messages, a malicious legislator could cause different legislators' ledgers to be inconsistent. Inconsistency could also result from a lapse of memory by an honest legislator or messenger.

When inconsistencies were recognized, they could easily be corrected by passing decrees. For example, disagreement about the current olive tax could be eliminated by passing a new decree declaring the tax to have a certain value. The difficult problem lay in correcting inconsistent ledgers even if no one was aware of the inconsistency.

The existence of dishonesty or mistakes by legislators can be inferred from the redundant decrees that began appearing in ledgers several years after the founding of Parliament. For example, the decree

2605: The olive tax is 9 drachmas per ton

was passed even though decree 2155 had already set the olive tax to 9 drachmas per ton, and no intervening decree had changed it. Parliament apparently cycled through its laws every six months so that even if legislators' ledgers were initially inconsistent, all legislators would agree on the current law of the land within six months. It is believed that by the use of these redundant decrees, the Paxons made their Parliament self-stabilizing. (*Self-stabilizing* is a modern term due to Dijkstra [1974].)

It is not clear precisely what self-stabilization meant in a Parliament with legislators coming and going at will. The Paxons would not have been satisfied with a definition that required all legislators to be in the Chamber at one time before consistency could be guaranteed. However, achieving consistency required that if one legislator had an entry in his ledger for a certain decree number and another did not, then the second legislator would eventually fill in that entry.

Unfortunately, we don't know exactly what sort of self-stabilization property the Paxon Parliament possessed or how it was achieved. Paxon mathematicians undoubtedly addressed the problem, but their work has not yet been found. I hope that future archaeological expeditions to Paxos will give high priority to the search for manuscripts on self-stabilization.

3.3.6 Choosing New Legislators

At first, membership in Parliament was hereditary, passing from parent to child. When the elder statesman Παρνασσις retired, he gave his ledger to his son, who carried on without interruption. It made no difference to other legislators which Παρνασσις they communicated with.

As old families emigrated and new ones immigrated, this system had to change. The Paxons decided to add and remove members of Parliament by decree. This posed a circularity problem: membership in Parliament was determined by which decrees were passed, but passing a decree required knowing what constituted a

majority set, which in turn depended upon who was a member of Parliament. The circularity was broken by letting the membership of Parliament used in passing decree n be specified by the law as of decree $n - 3$. A president could not try to pass decree 3255 until he knew all decrees through decree 3252. In practice, after passing the decree

3252: $\Sigma \tau \rho \omega \nu \gamma$ is now a legislator

the president would immediately pass the “olive-day” decree as decrees 3253 and 3254.

Changing the composition of Parliament in this way was dangerous and had to be done with care. The consistency and progress conditions would always hold. However, the progress condition guaranteed progress only if a majority set was in the Chamber; it did not guarantee that a majority set would ever be there. In fact, the mechanism for choosing legislators led to the downfall of the Parliamentary system in Paxos. Because of a scribe’s error, a decree that was supposed to honor sailors who had drowned in a shipwreck instead declared them to be the only members of Parliament. Its passage prevented any new decrees from being passed—including the decrees proposed to correct the mistake. Government in Paxos came to a halt. A general named $\Lambda\alpha\mu\pi\sigma\omega\nu$ took advantage of the confusion to stage a coup, establishing a military dictatorship that ended centuries of progressive government. Paxos grew weak under a series of corrupt dictators, and was unable to repel an invasion from the east that led to the destruction of its civilization.

4 Relevance to Computer Science

4.1 The State Machine Approach

Although Paxos’ Parliament was destroyed many centuries ago, its protocol is still useful. For example, consider a simple distributed database system that might be used as a name server. A state of the database consists of an assignment of values to names. Copies of the database are maintained by multiple servers. A client program can issue, to any server, a request to read or change the value assigned to a name. There are two kinds of read request: a *slow read*, which returns the value currently assigned to a name, and a *fast read*, which is faster but might not reflect a recent change to the database.

Table 1 shows the obvious correspondence between this database system and the Paxos Parliament: A client’s request to change a value is performed by passing a decree. A *slow read* involves passing a decree, as described in Section 3.3.4. A

Table 1

Parliament		Distributed Database
legislator	\leftrightarrow	server
citizen	\leftrightarrow	client program
current law	\leftrightarrow	database state

<i>command:</i>	read (<i>name, client</i>)	update (<i>name, val, client</i>)
<i>response:</i>	(<i>client, value of name</i>)	(<i>client, "ok"</i>)
<i>new state:</i>	Same as current state	Same as current state except value of <i>name</i> changed to <i>val</i>

Figure 2 State machine for simple database.

fast read is performed by reading the server’s current version of the database. The Paxon Parliament protocol provides a distributed, fault-tolerant implementation of the database system,

This method of implementing a distributed database is an instance of the state machine approach, first proposed in Lamport [1978]. In this approach, one first defines a *state machine*, which consists of a set of states, a set of commands, a set of responses, and a function that assigns a response/state pair (a pair consisting of a response and a state) to each command/state pair. Intuitively, a state machine executes a command by producing a response and changing its state; the command and the machine’s current state determine its response and its new state. For the distributed database, a state-machine state is just a database state. The state-machine commands and the function specifying the response and new state are described in Figure 2.

In the state machine approach, a system is implemented with a network of server processes. The servers transform client requests into state machine commands, execute the commands, and transform the state-machine responses into replies to clients. A general algorithm ensures that all servers obtain the same sequence of commands, thereby ensuring that they all produce the same sequence of responses and state changes—assuming they all start from the same initial state. In the database example, a client request to perform a *slow read* or to change a value is transformed into a state-machine **read** or **update** command. That command is executed, and the state-machine response is transformed into a reply to the client,

which is sent to him by the server who received his request. Since all servers perform the same sequence of state-machine commands, they all maintain consistent versions of the database. However, at any time, some servers may have earlier versions than others because a state-machine command need not be executed at the same time by all servers. A server uses his current version of the state to reply to a *fast read* request, without executing a state-machine command.

The functionality of the system is expressed by the state machine, which is just a function from command/state pairs to response/state pairs. Problems of synchronization and fault-tolerance are handled by the general algorithm with which servers obtain the sequence of commands. When designing a new system, only the state machine is new. The servers obtain the state-machine commands by a standard distributed algorithm that has already been proved correct. Functions are much easier to design, and to get right, than distributed algorithms.

The first algorithm for implementing an arbitrary state machine appeared in Lamport [1978]. Later, algorithms were devised to tolerate up to any fixed number f of arbitrary failures [Lamport 1984]. These algorithms guarantee that, if fewer than f processes fail, then state machine commands are executed within a fixed length of time. The algorithms are thus suitable for applications requiring real-time response.¹⁴ But if more than f failures occur, then different servers may have inconsistent copies of the state machine. Moreover, the inability of two servers to communicate with each other is equivalent to the failure of one of them. For a system to have a low probability of losing consistency, it must use an algorithm with a large value of f , which in turn implies a large cost in redundant hardware, communication bandwidth, and response time.

The Paxos Parliament's protocol provides another way to implement an arbitrary state machine. The legislators' law book corresponds to the machine state, and passing a decree corresponds to executing a state-machine command. The resulting algorithm is less robust and less expensive than the earlier algorithms. It does not tolerate arbitrary, malicious failures, nor does it guarantee bounded-time response. However, consistency is maintained despite the (benign) failure of any number of processes and communication paths. The Paxos algorithm is suitable for systems with modest reliability requirements that do not justify the expense of an extremely fault-tolerant, real-time implementation.

If the state machine is executed with an algorithm that guarantees bounded-time response, then time can be made part of the state, and machine actions can

14. These algorithms were derived from the military protocols of another Mediterranean state.

be triggered by the passage of time. For example, consider a system for granting ownership of resources. The state can include the time at which a client was granted a resource, and the state machine can automatically execute a command to revoke ownership if the client has held the resource too long.

With the Paxon algorithm, time cannot be made part of the state in such a natural way. If failures occur, it can take arbitrarily long to execute a command (pass a decree), and one command can be executed before (appear earlier in the sequence of decrees than) another command that was issued earlier. However, a state machine can still use real time the same way the Paxon Parliament did. For example, the method described in Section 3.3.3 for deciding who was the current cheese inspector can be used to decide who is the current owner of a resource.

4.2 Commit Protocols

The Paxon Synod protocol is similar to standard three-phase commit protocols [Bernstein 1987; Skeen 1982]. A Paxon ballot and a three-phase commit protocol both involve the exchange of five messages between a coordinator (the president) and the other quorum members (legislators). A commit protocol chooses one of two values—*commit* or *abort*—while the Synod protocol chooses an arbitrary decree. To convert a commit protocol to a Synod protocol, one sends the decree in the initial round of messages. A *commit* decision means that this decree was passed, and an *abort* decision means that the “olive-day” decree was passed.

The Synod protocol differs from a converted commit protocol because the decree is not sent until the second phase. This allows the corresponding parliamentary protocol to execute the first phase just once for all decrees, so the exchange of only three messages is needed to pass each individual decree.

The theorems on which the Synod protocol is based are similar to results obtained by Dwork, Lynch, and Stockmeyer [Dwork et al. 1988]. However, their algorithms execute ballots sequentially in separate rounds, and they seem to be unrelated to the Synod protocol.

Appendix: Proof of Consistency of the Synodic Protocol

A.1 The Basic Protocol

The Synod’s basic protocol, described informally in Section 2.3, is stated here using modern algorithmic notation. We begin with the variables that a priest p must maintain. First come the variables that represent information kept in his

Sidebar 2

Much research has been done in the field since this article was written. The state-machine approach has been surveyed by Schneider [1990]. The recovery protocol by Keidar and Dolev [1996] and the totally-ordered broadcast algorithm of Fekete et al. [1997] are quite similar to the Paxos protocol described here. The author was also apparently unaware that the view management protocol by Oki and Liskov [1988] seems to be equivalent to the Paxos protocol.

Many of the refinements presented in this submission have also appeared in contemporary or subsequent articles. The method of delegation described in Section 3.3.3 is very similar to the leases mechanism of Gray and Cheriton [1989]. The technique of Section 3.3.4 in which the Paxons satisfy the monotonicity condition by using decree numbers is described by Ladin et al. [1992]. The technique of Section 3.3.6 for adding new legislators was also given by Schneider [1990].

K. M.

ledger. (For convenience, the vote $prevVote[p]$ used in Section 2.3 is replaced by its components $prevBal[p]$ and $prevDec[p]$.)

- $outcome[p]$ The decree written in p 's ledger, or BLANK if there is nothing written there yet.
- $lastTried[p]$ The number of the last ballot that p tried to begin, or $-\infty$ if there was none.
- $prevBal[p]$ The number of the last ballot in which p voted, or $-\infty$ if he never voted.
- $prevDec[p]$ The decree for which p last voted, or BLANK if p never voted.
- $nextBal[p]$ The number of the last ballot in which p agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest p could keep on a slip of paper:

- $status[p]$ One of the following values:
 - $idle$ Not conducting or trying to begin a ballot
 - $trying$ Trying to begin ballot number $lastTried[p]$
 - $polling$ Now conducting ballot number $lastTried[p]$
- If p has lost his slip of paper, then $status[p]$ is assumed to equal $idle$ and the values of the following four variables are irrelevant.

$prevVotes[p]$	The set of votes received in <i>LastVote</i> messages for the current ballot (the one with ballot number $lastTried[p]$).
$quorum[p]$	If $status[p] = polling$, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.
$voters[p]$	If $status[p] = polling$, then the set of quorum members from whom p has received <i>Voted</i> messages in the current ballot; otherwise, meaningless.
$decree[p]$	If $status[p] = polling$, then the decree of the current ballot; otherwise, meaningless.

There is also the history variable \mathcal{B} , which is the set of ballots that have been started and their progress—namely, which priests have cast votes. (A history variable is one used in the development and proof of an algorithm, but not actually implemented.)

Next come the actions that priest p may take. These actions are assumed to be atomic, meaning that once an action is begun, it must be completed before priest p begins any other action. An action is described by an enabling condition and a list of effects. The enabling condition describes when the action can be performed; actions that receive a message are enabled whenever a messenger has arrived with the appropriate message. The list of effects describes how the action changes the algorithm’s variables and what message, if any, it sends. (Each individual action sends at most one message.)

Recall that ballot numbers were partitioned among the priests. For any ballot number b , the Paxons defined $owner(b)$ to be the priest who was allowed to use that ballot number.

The actions in the basic protocol are allowed actions; the protocol does not require that a priest ever do anything. No attempt at efficiency has been made; the actions allow p to do silly things, such as sending another *BeginBallot* message to a priest from whom he has already received a *LastVote* message.

Try New Ballot

Always enabled.

- Set $lastTried[p]$ to any ballot number b , greater than its previous value, such that $owner[b] = p$.
- Set $status[p]$ to *trying*.
- Set $prevVotes[p]$ to \emptyset .

Send NextBallot Message

Enabled whenever $status[p] = trying$.

- Send a *NextBallot*($lastTried[p]$) message to any priest.

Receive $NextBallot(b)$ Message

If $b \geq nextBal[p]$ then

- Set $nextBal[p]$ to b .

Send $LastVote$ Message

Enabled whenever $nextBal[p] > prevBal[p]$.

- Send a $LastVote(nextBal[p], v)$ message to priest $owner(nextBal[p])$, where $v_{pst} = p$, $v_{bal} = prevBal[p]$, and $v_{dec} = prevDec[p]$.

Receive $LastVote(b, v)$ Message

If $b = lastTried[p]$ and $status[p] = trying$, then

- Set $prevVotes[p]$ to the union of its original value and $\{v\}$.

Start Polling Majority Set Q

Enabled when $status[p] = trying$ and $Q \subseteq \{v_{pst} : v \in prevVotes[p]\}$, where Q is a majority set.

- Set $status[p]$ to *polling*.
- Set $quorum[p]$ to Q .
- Set $voters[p]$ to \emptyset .
- Set $decree[p]$ to a decree d chosen as follows: Let v be the maximum element of $prevVotes[p]$. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else d can equal any decree.
- Set \mathcal{B} to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and $B_{bal} = lastTried[p]$.

Send $BeginBallot$ Message

Enabled when $status[p] = polling$.

- Send a $BeginBallot(lastTried[p], decree[p])$ message to any priest in $quorum[p]$.

Receive $BeginBallot(b, d)$ Message

If $b = nextBal[p] > prevBal[p]$ then

- Set $prevBal[p]$ to b .
- Set $prevDec[p]$ to d .
- If there is a ballot B in \mathcal{B} with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of \mathcal{B} be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

Send Voted Message

Enabled whenever $prevBal[p] \neq -\infty$.

- Send a $Voted(prevBal[p], p)$ message to $ownerprevBal[p]$.

Receive Voted(b, q) Message

$b = lastTried[p]$ and $status[p] = polling$, then

- Set $voters[p]$ to the union of its old value and $\{q\}$.

Succeed

Enabled whenever $status[p] = polling$, $participants[p] \subseteq voters[p]$, and $outcome[p] = BLANK$.

- Set $outcome[p]$ to $decree[p]$.

Send Success Message

Enabled whenever $outcome[p] \neq BLANK$.

- Send a $Success(outcome[p])$ message to any priest.

Receive Success(d) Message

If $outcome[p] = BLANK$, then

- Set $outcome[p]$ to d .

This algorithm is an abstract description of the real protocol performed by Paxon priests. Do the algorithm's actions accurately model the actions of the real priests? There were three kinds of actions that a priest could perform "atomically": receiving a message, writing a note or ledger entry, and sending a message. Each of these is represented by a single action of the algorithm, except that **Receive** actions both receive a message and set a variable. We can pretend that the receipt of a message occurred when a priest acted upon the message; if he left the Chamber before acting upon it, then we can pretend that the message was never received. Since this pretense does not affect the consistency condition, we can infer the consistency of the basic Synod protocol from the consistency of the algorithm.

A.2 Proof of Consistency

To prove the consistency condition, it is necessary to show that whenever $outcome[p]$ and $outcome[q]$ are both different from $BLANK$, they are equal. A rigorous correctness proof requires a complete description of the algorithm. The description given above is almost complete. Missing is a variable \mathcal{M} whose value is the

multiset of all messages in transit.¹⁵ Each **Send** action adds a message to this multiset and each **Receive** action removes one. Also needed are actions to represent the loss and duplication of messages, as well as a **Forget** action that represents a priest losing his slip of paper.

With these additions, we get an algorithm that defines a set of possible behaviors, in which each change of state corresponds to one of the allowed actions. The Paxons proved correctness by finding a predicate I such that

1. I is true initially.
2. I implies the desired correctness condition.
3. Each allowed action leaves I true.

The predicate I was written as a conjunction $I1 \wedge \dots \wedge I7$, where $I1$ – $I5$ were in turn the conjunction of predicates $I1(p)$ – $I5(p)$ for all priests p . Although most variables are mentioned in several of the conjuncts, each variable except $status(p)$ is naturally associated with one conjunct, and each conjunct can be thought of as a constraint on its associated variables. The definitions of the individual conjuncts of I are given below, where a list of items marked by \wedge symbols denotes the conjunction of those items. The variables associated with a conjunct are listed in bracketed comments.

The Paxons had to prove that I satisfies the three conditions given above. The first condition, that I holds initially, requires checking that each conjunct is true for the initial values of all the variables. While not stated explicitly, these initial values can be inferred from the variables' descriptions, and checking the first condition is straightforward. The second condition, that I implies consistency, follows from $I1$, the first conjunct of $I6$, and Theorem 1. The hard part was proving the third condition, the invariance of I , which meant proving that I is left true by every action. This condition is proved by showing that, for each conjunct of I , executing any action when I is true leaves that conjunct true. The proofs are sketched below.

Proof Sketch for $I1(p)$ \mathcal{B} is changed only by adding a new ballot or adding a new priest to B_{vot} for some $B \in \mathcal{B}$, neither of which can falsify $I1(p)$. The value of $outcome[p]$ is changed only by the **Succeed** and **Receive Success Message** actions. The enabling condition and $I5(p)$ imply that $I1(p)$ is left true by the **Succeed** action. The enabling condition, $I1(p)$, and the last conjunct of $I7$ imply that $I1(p)$ is left true by the **Receive Success Message** action. ■

15. A multiset is a set that may contain multiple copies of the same element.

$I1(p) \triangleq$ [Associated variable: $outcome[p]$]
 $(outcome[p] \neq BLANK) \Rightarrow \exists B \in \mathcal{B}: (B_{qrm} \subseteq B_{vot}) \wedge (B_{dec} = outcome[p])$

$I2(p) \triangleq$ [Associated variable: $lastTried[p]$]
 $\wedge owner(lastTried[p]) = p$
 $\wedge \forall B \in \mathcal{B}: (owner(B_{bal}) = p) \Rightarrow$
 $\wedge B_{bal} \leq lastTried[p]$
 $\wedge (status[p] = trying) \Rightarrow (B_{bal} < lastTried[p])$

$I3(p) \triangleq$ [Associated variables: $prevBal[p]$, $prevDec[p]$, $nextBal[p]$]
 $\wedge prevBal[p] = MaxVote(\infty, p, \mathcal{B})_{bal}$
 $\wedge prevDec[p] = MaxVote(\infty, p, \mathcal{B})_{dec}$
 $\wedge nextBal[p] \geq prevBal[p]$

$I4(p) \triangleq$ [Associated variable: $prevVotes[p]$]
 $(status[p] \neq idle) \Rightarrow$
 $\forall v \in prevVotes[p]: \wedge v = MaxVote(lastTried[p], v_{pst}, \mathcal{B})$
 $\wedge nextBal[v_{pst}] \geq lastTried[p]$

$I5(p) \triangleq$ [Associated variables: $quorum[p]$, $voters[p]$, $decree[p]$]
 $(status[p] = polling) \Rightarrow$
 $\wedge quorum[p] \subseteq \{v_{pst} : v \in prevVotes[p]\}$
 $\wedge \exists B \in \mathcal{B}: \wedge quorum[p] = B_{qrm}$
 $\wedge decree[p] = B_{dec}$
 $\wedge voters[p] \subseteq B_{vot}$
 $\wedge lastTried[p] = B_{bal}$

$I6 \triangleq$ [Associated variable: \mathcal{B}]
 $\wedge B1(\mathcal{B}) \wedge B2(\mathcal{B}) \wedge B3(\mathcal{B})$
 $\wedge \forall B \in \mathcal{B}: B_{qrm}$ is a majority set

$I7 \triangleq$ [Associated variable: \mathcal{M}]
 $\wedge \forall NextBallot(b) \in \mathcal{M}: (b \leq lastTried[owner(b)])$
 $\wedge \forall LastVote(b, v) \in \mathcal{M}: \wedge v = MaxVote(b, v_{pst}, \mathcal{B})$
 $\wedge nextBal[v_{pst}] \geq b$
 $\wedge \forall BeginBallot(b, d) \in \mathcal{M}: \exists B \in \mathcal{B}: (B_{bal} = b) \wedge (B_{dec} = d)$
 $\wedge \forall Voted(b, p) \in \mathcal{M}: \exists B \in \mathcal{B}: (B_{bal} = b) \wedge (p \in B_{vot})$
 $\wedge \forall Success(d) \in \mathcal{M}: \exists p: outcome[p] = d \neq BLANK$

Figure 3 Individual conjuncts of predicate I .

Proof Sketch for $I2(p)$ This conjunct depends only on $lastTried[p]$, $status[p]$, and \mathcal{B} . Only the **Try New Ballot** action changes $lastTried[p]$, and only that action can set $status[p]$ to *trying*. Since the action increases $lastTried[p]$ to a value b with $owner(b) = p$, it leaves $I2(p)$ true. A completely new element is added to \mathcal{B} only by a **Start Polling** action; the first conjunct of $I2(p)$ and the specification of the action imply that adding this new element does not falsify the second conjunct of $I2(p)$. The only other way \mathcal{B} is changed is by adding a new priest to B_{vot} for some $B \in \mathcal{B}$, which does not affect $I2(p)$. ■

Proof Sketch for $I3(p)$ Since votes are never removed from \mathcal{B} , the only action that can change $MaxVote(\infty, p, \mathcal{B})$ is one that adds to \mathcal{B} a vote cast by p . Only a **Receive BeginBallot Message** action can do that, and only that action changes $prevBal[p]$ and $prevDec[p]$. The **BeginBallot** conjunct of $I7$ implies that this action actually does add a vote to \mathcal{B} , and $B1(\mathcal{B})$ (the first conjunct of $I6$) implies that there is only one ballot to which the vote can be added. The enabling condition, the assumption that $I3(p)$ holds before executing the action, and the definition of $MaxVote$ then imply that the action leaves the first two conjuncts of $I3(p)$ true. The third conjunct is left true because $prevBal[p]$ is changed only by setting it to $nextBal[p]$, and $nextBal[p]$ is never decreased. ■

Proof Sketch for $I4(p)$ This conjunct depends only upon the values of $status[p]$, $prevVotes[p]$, $lastTried[p]$, $nextBal[q]$ for some priests q , and \mathcal{B} . The value of $status[p]$ is changed from *idle* to not *idle* only by a **Try New Ballot** action, which sets $prevVotes[p]$ to \emptyset , making $I4(p)$ vacuously true. The only other actions that change $prevVotes[p]$ are the **Forget** action, which leaves $I4(p)$ true because it sets $status[p]$ to *idle*, and the **Receive LastVote Message** action. It follows from the enabling condition and the **LastVote** conjunct of $I7$ that the **Receive LastVote Message** action preserves $I4(p)$. The value of $lastTried[p]$ is changed only by the **Try New Ballot** action, which leaves $I4(p)$ true because it sets $status[p]$ to *trying*. The value of $nextBal[q]$ can only increase, which cannot make $I4(p)$ false. Finally, $MaxVotestLastTried[p]v_{pst}\mathcal{B}$ can be changed only if v_{pst} is added to B_{vot} for some $B \in \mathcal{B}$ with $B_{bal} < lastTried[p]$. But v_{pst} is added to B_{vot} (by a **Receive BeginBallot Message** action) only if $nextBalv_{pst} = B_{bal}$, in which case $I4(p)$ implies that $B_{bal} \geq lastTried[p]$. ■

Proof Sketch for $I5(p)$ The value of $status[p]$ is set to *polling* only by the **Start Polling** action. This action's enabling condition guarantees that the first conjunct becomes true, and it adds the ballot to \mathcal{B} that makes the second conjunct true. No other action changes $quorum[p]$, $decree[p]$, or $lastTried[p]$ while leaving $status[p]$ equal to *polling*. The value of $prevVotes[p]$ cannot be changed while $status[p] = polling$, and \mathcal{B} is changed only by adding new elements or by adding a new priest to B_{vot} . The only remaining possibility for falsifying $I5(p)$ is the addition of a new element to $voters[p]$ by the

Receive Voted Message action. The *Voted* conjunct of $I7, B1(\mathcal{B})$ (the first conjunct of $I6$), and the action's enabling condition imply that the element added to $voters[p]$ is in B_{vot} , where B is the ballot whose existence is asserted in $I5(p)$. ■

Proof Sketch for $I6$ Since B_{bal} and B_{qrm} are never changed for any $B \in \mathcal{B}$, the only way $B1(\mathcal{B})$, $B2(\mathcal{B})$, and the second conjunct of $I6$ can be falsified is by adding a new ballot to \mathcal{B} , which is done only by the **Start Polling Majority Set Q** action when $status[p]$ equals *trying*. It follows from the second conjunct of $I2(p)$ that this action leaves $B1(\mathcal{B})$ true; and the assertion, in the enabling condition, that Q is a majority set implies that the action leaves $B2(\mathcal{B})$ and the second conjunct of $I6$ true. There are two possible ways of falsifying $B3(\mathcal{B})$: changing $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ by adding a new vote to \mathcal{B} , and adding a new ballot to \mathcal{B} . A new vote is added only by the **Receive BeginBallot Message** action, and $I3(p)$ implies that the action adds a vote later than any other vote cast by p in \mathcal{B} , so it cannot change $MaxVote(B_{bal}, B_{qrm}, \mathcal{B})$ for any B in \mathcal{B} . Conjunct $I4(p)$ implies that the new ballot added by the **Start Polling** action does not falsify $B3(\mathcal{B})$. ■

Proof Sketch for $I7$ $I7$ can be falsified either by adding a new message to \mathcal{M} or by changing the value of another variable on which $I7$ depends. Since $lastTried[p]$ and $nextBal[p]$ are never decreased, changing them cannot make $I7$ false. Since $outcome[p]$ is never changed if its value is not *BLANK*, changing it cannot falsify $I7$. Since \mathcal{B} is changed only by adding ballots and adding votes, the only change to it that can make $I7$ false is the addition of a vote by v_{pst} that makes the $LastVote(b, v)$ conjunct false by changing $MaxVote(b, v_{pst}, \mathcal{B})$. This can happen only if v_{pst} votes in a ballot B with $B_{bal} < b$. But v_{pst} can vote only in ballot number $nextBal[v_{pst}]$, and the assumption that this conjunct holds initially implies that $nextBal[v_{pst}] \geq b$. Therefore, we need check only that every message that is sent satisfies the condition in the appropriate conjunct of $I7$.

NextBallot: Follows from the definition of the **Send NextBallot Message** action and the first conjunct of $I2(p)$.

LastVote: The enabling condition of the **Send LastVote Message** action and $I3(p)$ imply that $MaxVote(nextBal[p], p, \mathcal{B}) = MaxVote(\infty, p, \mathcal{B})$, from which it follows that the *LastVote* message sent by the action satisfies the condition in $I7$.

BeginBallot: Follows from $I5(p)$ and the definition of the **Send BeginBallot Message** action.

Voted: Follows from $I3(p)$, the definition of *MaxVote*, and the definition of the **Send Voted Message** action.

Success: Follows from the definition of **Send Success Message**. ■

Acknowledgments

Daniel Duchamp pointed out to me the need for a new state-machine implementation. Discussions with Martín Abadi, Andy Hisgen, Tim Mann, and Garret Swart led me to Paxos. Λεωνίδας Γκίμπας provided invaluable assistance with the Paxon dialect.

References

- Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.
- De Prisco, R., Lampson, B., and Lynch, N. 1997. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, M. Mavronicolas and P. Tsigas, Eds., Lecture Notes in Computer Science, vol. 1320. Springer-Verlag, Berlin, Germany, 111–125.
- Dijkstra, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11, 643–644.
- Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (Apr.), 288–323.
- Fekete, A., Lynch, N., and Shvartsman, A. 1997. Specifying and using a partitionable group communication service. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, New York, NY, 53–62.
- Fischer, M. J., Lynch, N. A., and Paterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 1 (Jan.), 374–382.
- Gray, C. and Cheriton, D. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (Dec. 3–6), 202–210.
- Keidar, I. and Dolev, D. 1996. Efficient message ordering in dynamic networks. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, New York, NY.
- Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.* 10, 4 (Nov.), 360–391.
- Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7, 558–565.
- Lamport, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.* 6, 2 (Apr.), 254–280.
- Lampson, B. W. 1996. How to build a highly available system using consensus. In *Distributed Algorithms*, O. Babaoglu and K. Marzullo, Eds. Springer Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Berlin, Germany, 1–17.
- Oki, B. M. and Liskov, B. H. 1988. Viewstamped replication: A general primary copy. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, August 15–17, 1988). ACM Press, New York, NY, 8–17.

- Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec.), 299–319.
- Skeen, M. D. 1982. Crash recovery in a distributed database system. Ph.D. thesis. University of California at Berkeley, Berkeley, CA.

References

- M. Abadi and L. Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284. DOI: [10.1016/0304-3975\(91\)90224-P](https://doi.org/10.1016/0304-3975(91)90224-P). 10, 113, 163
- M. Abadi and L. Lamport. 1993. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1): 73–132. DOI: [10.1145/151646.151649](https://doi.org/10.1145/151646.151649). 114
- M. Abadi and L. Lamport. 1994. Open systems in TLA. In J. H. Anderson, D. Peleg, and E. Borowsky (eds.), *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 81–90. ACM. DOI: [10.1145/197917.197960](https://doi.org/10.1145/197917.197960). 114
- M. Abadi and L. Lamport. 1995. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3): 507–534. DOI: [10.1145/203095.201069](https://doi.org/10.1145/203095.201069). 114
- J.-R. Abrial. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. 115
- J.-R. Abrial. 2010. *Modeling in Event-B*. Cambridge University Press. 115, 127
- S. V. Adve and M. D. Hill. 1990. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. pp. 2–14, 1990. DOI: [10.1145/325096.325100](https://doi.org/10.1145/325096.325100). 46
- Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. 1993. Atomic snapshots of shared memory. *J. ACM*, 40(4): 873–890. DOI: [10.1145/153724.153741](https://doi.org/10.1145/153724.153741). 64
- M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9: 37–49. DOI: [10.1007/BF01784241](https://doi.org/10.1007/BF01784241). 53
- B. Alpern and F. B. Schneider. 1985. Defining liveness. *Inf. Process. Lett.*, 21(4): 181–185. DOI: [10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0). 104
- P. A. Alberg and J. D. Day. 1976. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering, ICSE '76*, pp. 562–570. IEEE Computer Society Press. 83, 85
- American Mathematical Society. 2019a. AMS-LaTeX. <http://www.ams.org/publications/authors/tex/amslatex>; last retrieved 10 February 2019. 2, 145
- American Mathematical Society. 2019b. AMS-TeX. <http://www.ams.org/publications/authors/tex/amstex>; last retrieved 10 February 2019. 145

- K. R. Apt. 1981. Ten years of Hoare's logic: A survey—part 1. *ACM Trans. Prog. Lang. Syst.*, 3(4): 431–483, 1981. DOI: [10.1145/357146.357150](https://doi.org/10.1145/357146.357150). 109
- P. C. Attie, N. Francez, and O. Grumberg. 1993. Fairness and hyperfairness in multi-party interactions. *Distributed Computing*, 6(4): 245–254, 1993. DOI: [10.1007/BF02242712](https://doi.org/10.1007/BF02242712). 104
- H. Attiya, F. Ellen, and A. Morrison. 2015. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC '15*, pp. 385–394, New York. ACM. 53
- H. Attiya and J. Welch. 1998. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill International (UK), London, 1998. 37, 40
- H. Attiya and J. Welch. 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics, Second Edition*. John Wiley & Sons, Hoboken, NJ, 2004. 37
- H. Attiya and J. L. Welch. 1994. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2): 91–122. DOI: [10.1145/176575.176576](https://doi.org/10.1145/176575.176576). 46
- N. Azmy, S. Merz, and C. Weidenbach. 2018. A machine-checked correctness proof for pastry. *Sci. Comput. Program.*, 158: 64–80. DOI: [10.1016/j.scico.2017.08.003](https://doi.org/10.1016/j.scico.2017.08.003). 127
- R.-J. Back. 1981. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1): 49–68, 1981. DOI: [10.1016/0022-0000\(81\)90005-2](https://doi.org/10.1016/0022-0000(81)90005-2). 111
- J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern. 1957 The Fortran automatic coding system. In *Papers presented at the February 26–28, 1957, Western Joint Computer Conference: Techniques for Reliability*, pp. 188–198. ACM. DOI: [10.1145/1455567.1455599](https://doi.org/10.1145/1455567.1455599). 133
- C. Barrett and C. Tinelli. 2018. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem (eds.), *Handbook of Model Checking*, pp. 305–343. Springer. 126
- M. Ben-Or. 1983. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, PODC '83*, pp. 27–30. ACM. DOI: [10.1145/800221.806707](https://doi.org/10.1145/800221.806707). 90
- M. Ben-Or and R. El-Yaniv. 2003. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16: 249–262. DOI: [10.1007/s00446-002-0083-3](https://doi.org/10.1007/s00446-002-0083-3). 71
- D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn. 2015. SPHINCS: Practical stateless hash-based signatures. In *Proceedings EUROCRYPT 2015*, pp. 368–397. DOI: [10.1007/978-3-662-46800-5_15](https://doi.org/10.1007/978-3-662-46800-5_15). 81
- I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. 2016. Debugging distributed systems. *Queue*, 14(2): 50. 53
- K. Birman, A. Schiper, and P. Stephenson. August 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3): 272–314. DOI: [10.1145/128738.128742](https://doi.org/10.1145/128738.128742). 53, 102

- K. Birman. 1986. Isis: A system for fault-tolerant distributed computing. Technical report, Cornell University. [53](#)
- A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. September 1993. The Echo distributed file system. *Digital Systems Research Center Research Report 111*, 10. DOI: [10.1.1.43.1306](#). [148](#)
- R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. 2009. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pp. 4–10, Berkeley, CA. USENIX Association. [86](#)
- W. J. Bolosky, J. R. Douceur, and J. Howell. 2007. The Farsite project: A retrospective. *Operating Systems Review*, 41(2): 17–26. DOI: [10.1145/1243418.1243422](#). [128](#)
- R. Bonichon, D. Delahaye, and D. Doligez. 2007. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov (eds.), *14th International Conference Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4790 of *Lecture Notes in Computer Science*, pp. 151–165. Springer. DOI: [10.1007/978-3-540-75560-9_13](#). [126](#)
- F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. 2001. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technologies*, PaCT '01, pp. 42–50. Springer-Verlag. DOI: [10.1007/3-540-44743-1_4.pdf](#). [93](#)
- E. A. Brewer. July 2000. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, p. 7. DOI: [10.1145/343477.343502](#). [50](#)
- J. A. Buchmann, E. Dahmen, and A. Hülsing. 2011. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In *Proceedings Workshop on Post-Quantum Cryptography (PQC)*, volume 7071 of *Lecture Notes in Computer Science*, pp. 117–129. Springer. DOI: [10.1007/978-3-642-25405-5_8](#). [81](#)
- M. Burrows. January 2019. Personal communication. [150](#)
- M. Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 335–350. USENIX Association. [9](#), [100](#)
- R. M. Burstall. 1974. Program proving as hand simulation with a little induction. In *Information Processing*, pp. 308–312. North-Holland Pub. Co. [105](#)
- V. Bush. 1945. As we may think. *The Atlantic Monthly*, 176(1): pp. 101–108. [131](#)
- C. Cachin, R. Guerraoui, and L. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011. [54](#), [69](#)
- C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. August 2001. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference*, pp. 524–541. [71](#)
- S. Chand, Y. A. Liu, and S. D. Stoller. 2016. Formal verification of multi-Paxos for distributed consensus. In J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou (eds.), *21st*

- International Symposium Formal Methods (FM 2016)*, volume 9995 of *Lecture Notes in Computer Science*, pp. 119–136. DOI: [10.1007/978-3-319-48989-6_8](https://doi.org/10.1007/978-3-319-48989-6_8). 127
- T. D. Chandra, R. Griesemer, and J. Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC '07)*, pp. 398–407. ACM. DOI: [10.1145/1281100.1281103](https://doi.org/10.1145/1281100.1281103). 9, 99
- T. D. Chandra and S. Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2): 225–267. DOI: [10.1145/226643.226647](https://doi.org/10.1145/226643.226647). 71
- K. M. Chandy and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1): 63–75. DOI: [10.1145/214451.214456](https://doi.org/10.1145/214451.214456). 7, 48, 59, 64, 143, 164
- M. Chandy. July 2018. Personal communications (email), 16 and 30. 165
- A. Condon and A. J. Hu. 2003. Automatable verification of sequential consistency. *Theory Comput. Syst.*, 36(5): 431–460. DOI: [10.1007/s00224-003-1082-x](https://doi.org/10.1007/s00224-003-1082-x). 46
- D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. August 2012. TLA+ proofs. In D. Giannakopoulou and D. Méry (eds.), *FM 2012: Formal Methods - 18th International Symposium*, volume 7436 of *Lecture Notes in Computer Science*, pp. 147–154. Springer. 125
- G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *SOSP*. DOI: [10.1145/1323293.1294281](https://doi.org/10.1145/1323293.1294281). 53
- D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. 2018. Causal consistency and latency optimality: Friend or foe? *PVLDB*, 11(11): 1618–1632. DOI: [10.14778/3236187.3236210](https://doi.org/10.14778/3236187.3236210). 53
- W. Diffie and M. E. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6): 644–654. DOI: [10.1109/TIT.1976.1055638](https://doi.org/10.1109/TIT.1976.1055638). 78, 140
- E. W. Dijkstra. 1971. On the reliability of programs. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html>. 159
- E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *CACM*, 8(9): 569. 4
- E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Inf.*, 1: 115–138. DOI: [10.1007/BF00289519](https://doi.org/10.1007/BF00289519). 30, 44
- E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice Hall. 111
- D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. 1992. Protocol verification as a hardware design aid. In *IEEE International Conference Computer Design: VLSI in Computers and Processors*, pp. 522–525. IEEE Computer Society. 123
- D. Dolev and N. Shavit. 1997. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2): 418–455. DOI: [10.1137/S0097539790192647](https://doi.org/10.1137/S0097539790192647). 37, 40
- D. Dolev and H. R. Strong. 1982. Polynomial algorithms for multiple processor agreement. In *Proceedings Symposium on Theory of Computing (STOC)*, pp. 401–407. DOI: [10.1145/800070.802215](https://doi.org/10.1145/800070.802215). 76

- D. Dolev and H. R. Strong. 1983. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.* 12(4): 656–666. DOI: [10.1137/0212045](https://doi.org/10.1137/0212045). 76
- D. Dolev and A. C. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2): 198–207. DOI: [10.1109/TIT.1983.1056650](https://doi.org/10.1109/TIT.1983.1056650). 78
- D. Doligez, J. Kriener, L. Lamport, T. Libal, and S. Merz. 2014. Coalescing: Syntactic abstraction for reasoning in first-order modal logics. In C. Benzmüller and J. Otten (eds.), *Automated Reasoning in Quantified Non-Classical Logics*, volume 33 of *EPiC Series in Computing*, pp. 1–16. 126
- J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. 2014. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC ’14, pp. 4: 1–4: 13. ACM. DOI: [10.1145/2670979.2670983](https://doi.org/10.1145/2670979.2670983). 53
- C. Dwork, N. Lynch, and L. Stockmeyer. April 1988. Consensus in the presence of partial synchrony. *J. ACM*, 35(2): 288–323. DOI: [10.1145/42282.42283](https://doi.org/10.1145/42282.42283). 94
- C. Dwork and O. Waarts. 1992. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp. 655–666. DOI: [10.1145/129712.129776](https://doi.org/10.1145/129712.129776). 37, 40
- C. A. Ellis. 1977. Consistency and correctness of duplicate database systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP ’77, pp. 67–84. ACM. DOI: [10.1145/1067625.806548](https://doi.org/10.1145/1067625.806548). 84
- U. Engberg. 1996. *Reasoning in the Temporal Logic of Actions*. BRICS Dissertation Series. 151
- U. Engberg, P. Grønning, and L. Lamport. 1992. Mechanical verification of concurrent systems with TLA. In G. von Bochmann and D. K. Probst (eds.), *Computer Aided Verification, Fourth International Workshop*, CAV ’92, volume 663 of *Lecture Notes in Computer Science*, pp. 44–55. Springer. DOI: [10.1007/3-540-56496-9_5](https://doi.org/10.1007/3-540-56496-9_5). 115
- C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. Dissertation, Australian National University. 53
- M. J. Fischer, N. A. Lynch, and M. S. Paterson. April 1985. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2): 374–382. DOI: [10.1145/3149.214121](https://doi.org/10.1145/3149.214121). 89, 148
- M. Fitzi and J. A. Garay. 2003. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 211–220. DOI: [10.1145/872035.872066](https://doi.org/10.1145/872035.872066). 70
- R. W. Floyd. 1967. Assigning meanings to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, pp. 19–32. American Mathematical Society. 105
- R. Friedman, A. Mostéfaoui, S. Rajsbaum, and M. Raynal. 2007. Asynchronous agreement and its relation with error-correcting codes. *IEEE Trans. Computers*, 56(7): 865–875. DOI: [10.1109/TC.2007.1043](https://doi.org/10.1109/TC.2007.1043). 72
- E. Gafni and L. Lamport. 2003. Disk Paxos. *Distributed Computing*, 16(1): 1–20. DOI: [10.1007/s00446-002-0070-8](https://doi.org/10.1007/s00446-002-0070-8). 9, 100, 161

- J. A. Garay and Y. Moses. 1998. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. 27(1): 247–290. DOI: [10.1137/S0097539794265232](https://doi.org/10.1137/S0097539794265232). 76
- S. J. Garland and J. V. Guttag. 1988. LP: The Larch prover. In *International Conference on Automated Deduction*, pp. 748–749. Springer. DOI: [10.1007/BFb0012879](https://doi.org/10.1007/BFb0012879). 151
- S. Ghemawat, H. Gobioff, and S.-T. Leung. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43. ACM. DOI: [10.1145/1165389.945450](https://doi.org/10.1145/1165389.945450). 101
- P. B. Gibbons and E. Korach. 1997. Testing shared memories. *SIAM J. Comput.*, 26(4): 1208–1244. DOI: [10.1.1.107.3013](https://doi.org/10.1.1.107.3013). 46
- P. B. Gibbons, M. Merritt, and K. Gharachorloo. 1991. Proving sequential consistency of high-performance shared memories (extended abstract). In *SPAA*, pp. 292–303. DOI: [10.1145/113379.113406](https://doi.org/10.1145/113379.113406). 46
- S. Gilbert and N. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2): 51–59. DOI: [10.1145/564585.564601](https://doi.org/10.1145/564585.564601). 50
- E. Gilkerson and L. Lamport. 2004. On hair color in France. *Annals of Improbable Research*, pp. 18–19. 165
- P. Godefroid and P. Wolper. 1994. A partial approach to model checking. *Inf. Comput.*, 110(2): 305–326. DOI: [10.1006/inco.1994.1035](https://doi.org/10.1006/inco.1994.1035). 127
- C. Gray and D. Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pp. 202–210. ACM. DOI: [10.1145/74851.74870](https://doi.org/10.1145/74851.74870). 99
- J. Gray and L. Lamport. 2006. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1): 133–160, 2006. DOI: [10.1145/1132863.1132867](https://doi.org/10.1145/1132863.1132867). 162
- D. Gries and F. B. Schneider. 1995. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pp. 366–373. Springer. 116
- Y. Gurevich. 1995. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and validation methods*, pp. 9–36. Oxford University Press. 127
- C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. 2015. Ironfleet: Proving practical distributed systems correct. In E. L. Miller and S. Hand (eds.), *Proceedings 25th Symposium Operating Systems Principles*, SOSP 2015, pp. 1–17. ACM. DOI: [10.1145/2815400.2815428](https://doi.org/10.1145/2815400.2815428). 128
- C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. 2017. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7): 83–92. DOI: [10.1145/3068608](https://doi.org/10.1145/3068608). 128
- M. Herlihy. January 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1): 124–149. DOI: [10.1145/114005.102808](https://doi.org/10.1145/114005.102808). 4
- M. Herlihy and J. M. Wing. July 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3): 463–492. DOI: [10.1145/78969.78972](https://doi.org/10.1145/78969.78972). 5, 71

- A. Hisgen, A. Birrell, T. Mann, M. Schroeder, and G. Swart. September 1989. Availability and consistency tradeoffs in the Echo distributed file system. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pp. 49–54. IEEE Computer Society. DOI: [10.1109/WWOS.1989.109267](https://doi.org/10.1109/WWOS.1989.109267). 96
- G. J. Holzmann. 2003. *The SPIN Model Checker*. Addison-Wesley. 123
- H. Howard, D. Malkhi, and A. Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*. 162
- P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. 2010. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*. 9, 100
- IBM. 1955. *IBM 705 EDPM Electronic Data Processing Machine*. Internation Business Machines Corp. 133
- A. Israeli and M. Li. 1993. Bounded time-stamps. *Distributed Computing*, 6(4): 205–209. DOI: [10.1007/BF02242708](https://doi.org/10.1007/BF02242708). 37
- A. Israeli and A. Shaham. 2005. Time and space optimal implementations of atomic multi-writer register. *Inf. Comput.*, 200(1): 62–106. 40
- K. E. Iverson. 1962. A programming language. In *Proceedings of the Spring Joint Computer Conference*, pp. 345–351. ACM. 135
- P. Jayanti. 2005. An optimal multi-writer snapshot algorithm. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pp. 723–732. ACM. DOI: [10.1145/1060590.1060697](https://doi.org/10.1145/1060590.1060697). 64
- J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. 2007. Formal specification of a web services protocol. *J. Log. Algebr. Program.*, 70(1): 34–52. DOI: [10.1016/j.jlap.2006.05.004](https://doi.org/10.1016/j.jlap.2006.05.004). 128
- P. R. Johnson and R. H. Thomas. 1975. The maintenance of duplicate databases. Network Working Group RFC 677. 6, 50, 53, 83, 137
- R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. R. Tuttle, and Y. Yu. 2003. Checking cache-coherence protocols with TLA⁺. *Formal Methods in System Design*, 22(2): 125–131. DOI: [10.1023/A:1022969405325](https://doi.org/10.1023/A:1022969405325). 123, 127
- R. Koo and S. Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, (1): 23–31. DOI: [10.1109/TSE.1987.232562](https://doi.org/10.1109/TSE.1987.232562). 64
- F. Kröger. 1977. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8: 243–266. DOI: [10.1007/BF00264469](https://doi.org/10.1007/BF00264469). 105
- DEC WRL. October 2018. MultiTitan: Four architecture papers. <http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-87-8.pdf>. 147
- P. B. Ladkin, L. Lamport, B. Olivier, and D. Roegel. 1999. Lazy caching in TLA. *Distributed Computing*, 12(2–3): 151–174. DOI: [10.1007/s004460050063](https://doi.org/10.1007/s004460050063). 155
- A. Lakshman and P. Malik. 2009. Cassandra: A decentralized structured storage system. In *SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS 2009)*. ACM. 100

- L. Lamport. 1957. Braid theory. *Mathematics Bulletin of the Bronx High School of Science* (1957), pp. 6,7,9. [132](#)
- L. Lamport. 1987. Distribution. Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87. [163](#)
- L. Lamport. 2019. My writings. [3, 29, 67, 103, 132, 134, 135, 141, 142, 148, 150, 151, 158, 165](#)
- L. Lamport. 2018. Personal communication (recorded conversation). [144, 145](#)
- L. Lamport. 1970. Comment on Bell's quadratic quotient method for hash coded searching. *Commun. ACM*, 13(9): 573–574. DOI: [10.1145/362736.362765](#). [135](#)
- L. Lamport. 1973. The coordinate method for the parallel execution of DO loops. In *Proceedings 1973 Sagamore Comput. Conf.*
- L. Lamport. 1974. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8): 453–455. DOI: [10.1145/361082.361093](#). [3, 29, 30, 35, 136](#)
- L. Lamport. 1974. The parallel execution of DO loops. *Commun. ACM*, 17(2): 83–93. DOI: [10.1145/360827.360844](#). [135](#)
- L. Lamport. 1977. Concurrent reading and writing. *Commun. ACM*, 20(11): 806–811. DOI: [10.1145/359863.359878](#). [30, 36, 44](#)
- L. Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2): 125–143. DOI: [10.1109/TSE.1977.229904](#). [30, 103, 104, 105, 106, 150](#)
- L. Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2: 95–114. DOI: [10.1016/0376-5075\(78\)90045-4](#). [84](#)
- L. Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7): 558–565. DOI: [10.1145/359545.359563](#). [6, 48, 58, 103, 137, 163, 164](#)
- L. Lamport. 1979a. Constructing digital signatures from a one way function. *Technical Report CSL-98, Computer Science Laboratory, SRI International*. [68, 78](#)
- L. Lamport. 1979b. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9): 690–691. DOI: [10.1109/TC.1979.1675439](#). [5, 30, 44, 143, 150](#)
- L. Lamport. 1979c. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1): 84–97. DOI: [10.1145/357062.357068](#). [103](#)
- L. Lamport. 1980. “Sometime” is sometimes “not never”—on the temporal logic of programs. In P. W. Abrahams, R. J. Lipton, and S. R. Bourne (eds.), *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 174–185. ACM Press. DOI: [10.1145/567446.567463](#). [106](#)
- L. Lamport. 1981. Password authentication with insecure communication. *Commun. ACM*, 24(11): 770–772. DOI: [10.1145/358790.358797](#). [68](#)
- L. Lamport. 1983. What good is temporal logic? In *IFIP Congress*, pp. 657–668. [111](#)
- L. Lamport. 1986a. The mutual exclusion problem: Part I: a theory of interprocess communication. *J. ACM*, 33(2): 313–326. DOI: [10.1145/5383.5384](#). [150](#)
- L. Lamport. 1986b. The mutual exclusion problem: Part II: statement and solutions. *J. ACM*, 33(2): 327–348. DOI: [10.1145/5383.5385](#). [150](#)

- L. Lamport. 1986c. On interprocess communication. Part I: basic formalism. *Distributed Computing*, 1(2): 77–85. DOI: [10.1007/BF01786227](https://doi.org/10.1007/BF01786227). [4](#), [29](#), [41](#), [42](#), [150](#)
- L. Lamport. 1986d. On interprocess communication. Part II: algorithms. *Distributed Computing*, 1(2): 86–101. DOI: [10.1007/BF01786228](https://doi.org/10.1007/BF01786228). [4](#), [29](#), [36](#), [37](#), [38](#), [39](#), [40](#), [42](#), [150](#)
- L. Lamport. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1): 1–11. DOI: [10.1145/7351.7352](https://doi.org/10.1145/7351.7352). [147](#)
- L. Lamport. 1990. *win* and *sin*: Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3): 396–428. DOI: [10.1145/78969.78970](https://doi.org/10.1145/78969.78970). [104](#)
- L. Lamport. 1991. The temporal logic of actions. *Research Report 79*, DEC Systems Research Center. [107](#)
- L. Lamport. 1992. Hybrid systems in TLA^+ . In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel (eds.), *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pp. 77–102. Springer. DOI: [10.1007/3-540-57318-6_25](https://doi.org/10.1007/3-540-57318-6_25). [115](#)
- L. Lamport. June 1993. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg (eds.), *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pp. 347–374. Springer. [151](#)
- L. Lamport. 1994. *LaTeX—A Document Preparation System: User's Guide and Reference Manual, Second Edition*. Pearson/Prentice Hall. [11](#), [145](#)
- L. Lamport. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3): 872–923. DOI: [10.1145/177492.177726](https://doi.org/10.1145/177492.177726). [107](#), [109](#), [151](#)
- L. Lamport. 1995. How to write a proof. *American Mathematical Monthly*, 102(7): 600–608. [125](#), [152](#), [154](#)
- L. Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169. DOI: [10.1145/279227.279229](https://doi.org/10.1145/279227.279229). [9](#), [54](#), [96](#), [163](#)
- L. Lamport. 1999. Specifying concurrent systems with TLA^+ . *Calculational System Design*. DOI: [10.1007%2F3-540-48153-2_6.pdf](https://doi.org/10.1007%2F3-540-48153-2_6.pdf). [152](#), [156](#)
- L. Lamport. 2000. Fairness and hyperfairness. *Distributed Computing*, 13(4): 239–245. DOI: [10.1007/PL00008921](https://doi.org/10.1007/PL00008921). [104](#)
- L. Lamport. 2000. How (La)TeX changed the face of mathematics. E-interview in DMV-Mitteilungen.
- L. Lamport. 2001. Paxos made simple. *SIGACT News*, 32(4): 51–58. [98](#), [150](#)
- L. Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. [10](#), [115](#), [152](#), [156](#), [163](#)
- L. Lamport. 2002. A discussion with Leslie Lamport. *IEEE Distributed Systems Online* 3, 8.
- L. Lamport. 2006c. Checking a multithreaded algorithm with ^+CAL . In S. Dolev, editor, *Distributed Computing, 20th International Symposium*, volume 4167 of *Lecture Notes in Computer Science*, pp. 151–163. Springer. DOI: [10.1007/11864219_11](https://doi.org/10.1007/11864219_11). [159](#)

- L. Lamport. 2006b. Fast Paxos. *Distributed Computing*, 19(2): 79–103. DOI: [10.1007/s00446-006-0005-x](https://doi.org/10.1007/s00446-006-0005-x). [161](#)
- L. Lamport. 2006. Measuring celebrity. *Annals of Improbable Research*, pp. 14–15. [165](#)
- L. Lamport. 2009. The PlusCal algorithm language. In M. Leucker and C. Morgan (eds.), *Theoretical Aspects of Computing*, volume 5684 of *Lecture Notes in Computer Science*, pp. 36–60. Springer. DOI: [10.1007/978-3-642-03466-4_2](https://doi.org/10.1007/978-3-642-03466-4_2). [10, 120, 159](#)
- L. Lamport. 2011. Byzantizing Paxos by refinement. In D. Peleg, editor, *Distributed Computing - 25th International Symposium*, volume 6950 of *Lecture Notes in Computer Science*, pp. 211–224. Springer. [127, 162](#)
- L. Lamport. 2012. Buridan’s principle. *Foundations of Physics*, 42(8): 1056–1066. DOI: [10.1007/s10701-012-9647-7](https://doi.org/10.1007/s10701-012-9647-7). [142](#)
- L. Lamport. 2012. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, March 6, 2012. DOI: [10.1007/s11784-012-0071-6](https://doi.org/10.1007/s11784-012-0071-6). [163](#)
- L. Lamport. 2014. TLA⁺: A preliminary guide.
- L. Lamport. 2015. The TLA⁺ hyperbook. [115](#)
- L. Lamport. 2015a. Turing lecture: The computer science of concurrency: the early years. *Commun. ACM*, 58(6): 71–76. DOI: [10.1145/2771951](https://doi.org/10.1145/2771951).
- L. Lamport. 2018. The TLA⁺ video course. [115](#)
- L. Lamport. August 2018b. Personal communication (email). [131, 132](#)
- L. Lamport. 2018c. Personal communication (email), 9 October 2018. [152](#)
- L. Lamport. 2019. TLA+ tools. <http://lamport.azurewebsites.net/tla/tools.html>; last retrieved 11 February 2019. [160](#)
- L. Lamport and R. Levin. 2016. Lamport, Leslie. Oral history, part 1. <https://www.computerhistory.org/collections/catalog/102717182>. [2, 132, 133, 134, 135, 136, 137, 138, 139, 141, 142, 146, 148, 149, 159, 163](#)
- L. Lamport and R. Levin. 2016. Lamport, Leslie. Oral history, part 2. <https://www.computerhistory.org/collections/catalog/102717246>. [169](#)
- L. Lamport, D. Malkhi, and L. Zhou. 2009a. Vertical Paxos and primary-backup replication. In S. Tirthapura and L. Alvisi (eds.), *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing*, pp. 312–313. ACM. DOI: [10.1145/1582716](https://doi.org/10.1145/1582716). [1582783, 9, 101, 162](#)
- L. Lamport, D. Malkhi, and L. Zhou. April 2009b. Stoppable Paxos. [162](#)
- L. Lamport, D. Malkhi, and L. Zhou. 2010. Reconfiguring a state machine. *SIGACT News*, 41(1): 63–73. DOI: [10.1145/1753171.1753191](https://doi.org/10.1145/1753171.1753191). [9, 102, 162](#)
- L. Lamport and M. Massa. 2004. Cheap Paxos. In *2004 International Conference on Dependable Systems and Networks*, pp. 307–314. IEEE Computer Society. [9, 101, 161](#)
- L. Lamport, J. Matthews, M. R. Tuttle, and Y. Yu. 2002. Specifying and verifying systems with TLA⁺. In G. Muller and E. Jul (eds.), *Proceedings of the 10th ACM SIGOPS European Workshop*, pp. 45–48. ACM. [153, 154, 157](#)

- L. Lamport and P. M. Melliar-Smith. 1985. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1): 52–78. DOI: [10.1145/2455.2457](https://doi.org/10.1145/2455.2457). [8](#), [76](#)
- L. Lamport and S. Merz. 2017. Auxiliary variables in TLA+. *CoRR*, abs/1703.05121. [113](#)
- L. Lamport and L. C. Paulson. 1999. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21(3): 502–526. DOI: [10.1145/319301.319317](https://doi.org/10.1145/319301.319317). [116](#)
- L. Lamport, R. E. Shostak, and M. C. Pease. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3): 382–401. DOI: [10.1145/357172.357176](https://doi.org/10.1145/357172.357176). [7](#), [8](#), [67](#), [68](#), [69](#), [70](#), [72](#), [73](#), [75](#), [76](#), [105](#), [141](#), [163](#)
- L. Lamport. 1972. *The Analytic Cauchy Problem with Singular Data*. Ph.D. thesis, Brandeis University.
- B. W. Lampson. 1996. How to build a highly available system using consensus. In *Workshop on Distributed Algorithms (WDAG)*, pp. 1–17. Springer. DOI: [10.1007/3-540-61769-8_1](https://doi.org/10.1007/3-540-61769-8_1). [9](#)
- B. W. Lampson. 2018. Personal communication (email). [139](#), [140](#), [149](#), [150](#), [167](#)
- B. W. Lampson and H. E. Sturgis. 1979. Crash recovery in a distributed storage system. Unpublished manuscript. [162](#), [167](#)
- D. Langworthy. 2005. Personal communication (email). [158](#)
- The L^AT_EX Project. 2019. <https://www.latex-project.org/>; last retrieved 6 May 2019. [145](#)
- E. K. Lee and C. A. Thekkath. 1996. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices*, volume 31, pp. 84–92. ACM. DOI: [10.1145/248209.237157](https://doi.org/10.1145/248209.237157). [100](#), [149](#)
- K. R. M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov (eds.), *16th International Conference Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer. DOI: [10.1007/978-3-642-17511-4_20](https://doi.org/10.1007/978-3-642-17511-4_20). [128](#)
- O. Lichtenstein and A. Pnueli. 1985. Checking that finite state concurrent programs satisfy their linear specification. In M. S. V. Deussen, Z. Galil, and B. K. Reid (eds.), *Proceedings Twelfth Ann. ACM Symposium Princ. Prog. Lang. (POPL 1985)*, pp. 97–107. ACM. DOI: [10.1145/318593.318622](https://doi.org/10.1145/318593.318622). [124](#)
- R. Lipton and J. Sandberg. 1988. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Computer Science Department, Princeton University. [46](#)
- B. Liskov and J. Cowling. 2012. Viewstamped replication revisited. 2012. [54](#)
- B. Liskov and R. Ladin. 1986. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC '86*, pp. 29–39. ACM. DOI: [10.1145/10590.10593](https://doi.org/10.1145/10590.10593). [53](#)
- W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, pp. 401–416, New York, NY. ACM. DOI: [10.1145/2043556.2043593](https://doi.org/10.1145/2043556.2043593). [53](#)

- W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 313–328. USENIX. 53
- N. A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann, San Francisco. 76
- D. Malkhi. 2018. Personal communication (recorded conversatino). 168
- F. Mattern. 1989. Virtual time and global states of distributed systems. In M. Cosnard et al. (eds.), *Proceedings of the International Workshop on Parallel Algorithms*. Elsevier. 53, 64
- K. McMillan. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers. 123
- R. C. Merkle. 1980. Protocols for public key cryptosystems. In *Proceedings IEEE Symposium on Security and Privacy*, pp. 122–134. DOI: [10.1109/SP.1980.10006](https://doi.org/10.1109/SP.1980.10006). 80
- S. Merz. 1999. A more complete TLA. In J. Wing, J. Woodcock, and J. Davies (eds.), *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pp. 1226–1244. Springer. DOI: [10.1007/3-540-48118-4_15](https://doi.org/10.1007/3-540-48118-4_15). 110
- S. Merz. 2008. The specification language TLA⁺. In D. Bjørner and M. C. Henson (eds.), *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pp. 401–451. Springer, Berlin-Heidelberg. DOI: [10.1007/978-3-540-74107-7_8](https://doi.org/10.1007/978-3-540-74107-7_8). 115, 160
- S. Merz. July 2018. Personal communication (email). 167
- Microsoft Research. 2019. *TLA⁺ Proof System*. <https://tla.msr-inria.inria.fr/tlaps/content/Home.html>; last retrieved 11 February 2019. 159
- D. Milojicic. 2002. A discussion with Leslie Lamport. *IEEE Distributed Systems Online* 3, 8. <https://www.microsoft.com/en-us/research/publication/discussion-leslie-lamport/>; last retrieved 6 May 2019. 161
- H. Minkowski. 2017. Translation:space and time—wikisource. [Online; accessed 15 March 2019]. 138
- C. Morgan. 1990. *Programming from specifications*. Prentice Hall. 111
- A. Mullery. December 1971. The distributed control of multiple copies of data. Technical Report RC 3642, IBM, Yorktown Heights, New York. 83
- M. Naor and M. Yung. 1989. Universal one-way hash functions and their cryptographic applications. In *Proceedings Symposium on Theory of Computing (STOC)*, pp. 33–43. DOI: [10.1145/73007.73011](https://doi.org/10.1145/73007.73011). 80
- National Institute of Standards and Technology. 2018. Post-quantum cryptography. Available at <https://csrc.nist.gov/projects/post-quantum-cryptography/>. 81
- C. Newcombe. 2012. Post on TLA+ discussion group. [https://groups.google.com/forum/#!searchin/tlaplus/professional\\$20career/tlaplus/ZJCi-UF31fc/Mawwi6U1CYJ](https://groups.google.com/forum/#!searchin/tlaplus/professional$20career/tlaplus/ZJCi-UF31fc/Mawwi6U1CYJ) 160
- C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. 2015. How Amazon web services uses formal methods. *CACM*, 58(4): 66–73. DOI: [10.1145/2699417](https://doi.org/10.1145/2699417). 128, 161

- M. J. K. Nielsen. Titan system manual. <http://www;hpl.hp.com/techreports/Compaq-DEC/WRL-86-1.pdf>, last retrieved 15 October 2018. [147](#)
- B. M. Oki and B. H. Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 8–17, ACM. DOI: [10.1145/62546.62549](https://doi.org/10.1145/62546.62549). [99, 149](#)
- D. Ongaro and J. Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings USENIX Annual Technical Conference ATC*. [54, 100](#)
- S. S. Owicki and D. Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5): 279–285. DOI: [10.1145/360051.360224](https://doi.org/10.1145/360051.360224). [30, 44](#)
- S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3): 455–495, 1982. DOI: [10.1145/357172.357178](https://doi.org/10.1145/357172.357178). [106, 107, 108](#)
- D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, (3): 240–247. DOI: [10.1109/TSE.1983.236733](https://doi.org/10.1109/TSE.1983.236733). [53](#)
- L. C. Paulson. 1994. *Isabelle: A Generic Theorem Prover*, volume 828 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg. See also the Isabelle home page at <http://isabelle.in.tum.de/>. [126](#)
- M. C. Pease, R. E. Shostak, and L. Lamport. 1980. Reaching agreement in the presence of faults. *J. ACM*, 27(2): 228–234. DOI: [10.1145/322186.3222188](https://doi.org/10.1145/322186.3222188). [7, 8, 67, 68, 75, 141, 163](#)
- A. Pnueli. 1977. The temporal logic of programs. In *Proceedings 18th Annual Symposium on the Foundations of Computer Science*, pp. 46–57. IEEE. DOI: [10.1109/SFCS.1977.32](https://doi.org/10.1109/SFCS.1977.32). [105](#)
- R. Prakash and M. Singhal. 1996. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1035–1048. DOI: [10.1109/71.539735](https://doi.org/10.1109/71.539735). [64](#)
- A. N. Prior. 1967. *Past, Present and Future*. Clarendon Press, Oxford, U.K. [105](#)
- Riak. Riak KV. <http://basho.com/products/riak-kv>. [53](#)
- R. L. Rivest, A. Shamir, and L. M. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2): 120–126. DOI: [10.1145/359340.359342](https://doi.org/10.1145/359340.359342). [80](#)
- T. R. Rodeheffer. 2018. Personal communication (recorded conversation and follow-up email), 9 October 2018 and 11 February 2019. [167](#)
- J. Rompel. 1990. One-way functions are necessary and sufficient for secure signatures. In *Proceedings Symposium on Theory of Computing (STOC)*, pp. 387–394. DOI: [10.1145/100216.100269](https://doi.org/10.1145/100216.100269). [80](#)
- J. H. Saltzer, D. P. Reed, and D. D. Clark. November 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4): 277–288. DOI: [10.1145/357401.357402](https://doi.org/10.1145/357401.357402). [86](#)

- M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4): 447–459. DOI: [10.1109/12.54838](https://doi.org/10.1109/12.54838). 53
- F. B. Schmuck. 1988. The use of efficient broadcast protocols in asynchronous distributed systems. Ph.D. thesis, Cornell University. 53
- F. Schneider. August 2018. Personal communication (email). 166
- F. B. Schneider. 1982. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst.*, 4(2): 125–148. DOI: [10.1145/357162.357163](https://doi.org/10.1145/357162.357163). 139
- F. B. Schneider. December 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4): 299–319. DOI: [10.1145/98163.98167](https://doi.org/10.1145/98163.98167). 86, 139
- R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. 1984. An interval-based temporal logic. In E. M. Clarke and D. Kozen (eds.), In *Proceedings Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pp. 443–457. Springer. 106
- R. Shostak. July 2018. Personal communication (email), 6–8. 140
- A. K. Singh, J. H. Anderson, and M. G. Gouda. 1994. The elusive atomic register. *J. ACM*, 41(2): 311–339. 37
- J. M. Spivey. 1992. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2nd edition. 115
- M. Suda and C. Weidenbach. 2012. A PLTL-prover based on labelled superposition with partial model guidance. In B. Gramlich, D. Miller, and U. Sattler (eds.), *6th International Joint Conference Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pp. 537–543. Springer. DOI: [10.1007/978-3-642-31365-3_42](https://doi.org/10.1007/978-3-642-31365-3_42). 126
- D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 172–183. DOI: [10.1145/224057.224070](https://doi.org/10.1145/224057.224070). 53
- A. Valmari. June 1990. A stubborn attack on state explosion. In *2nd International Wsh. Computer Aided Verification*, volume 531 of *LNCS*, pp. 156–165, Rutgers. Springer. DOI: [10.1007/BF00709154](https://doi.org/10.1007/BF00709154). 127
- R. van Renesse and F. B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*. USENIX Association. 84, 101
- K. Vidyasankar. 1988. Converting Lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6): 287–290. DOI: [10.1016/0020-0190\(88\)90175-5](https://doi.org/10.1016/0020-0190(88)90175-5). 39
- P. M. B. Vitányi and B. Awerbuch. 1986. Atomic shared register access by asynchronous hardware (detailed abstract). In *27th Annual Symposium on Foundations of Computer Science*, pp. 233–243. 40
- W. Vogels. 2009. Eventually consistent. *Commun. ACM*, 52(1): 40–44. DOI: [10.1145/1435417.1435432](https://doi.org/10.1145/1435417.1435432). 83

- P. Voldemort. Voldemort. <https://www.project-voldemort.com/voldemort>. 53
- S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. pp. 307–320. 100
- J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. 1978. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10): 1240–1255. DOI: [10.1109/PROC.1978.11114](https://doi.org/10.1109/PROC.1978.11114). 140, 164
- J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. 1978. Synchronizing clocks in the presence of faults. *Proceedings of the IEEE*, 66(10): 1240–1255. 67
- Wikipedia contributors. 2018a. Bravo (software)—Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Bravo_\(software\)](https://en.wikipedia.org/w/index.php?title=Bravo_(software)). [Online; accessed 25 February 2019]. 143
- Wikipedia contributors. 2018b. Eckhard Pfeiffer—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Eckhard_Pfeiffer, 2018. [Online; accessed 26 February 2019]. 155
- Wikipedia contributors. 2018c. Scribe (markup language)—Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Scribe_\(markup_language\)](https://en.wikipedia.org/w/index.php?title=Scribe_(markup_language)), 2018. [Online; accessed 25 February 2019]. 143
- Wikipedia contributors. 2018d. UNIVAC—Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=UNIVAC>, 2018. [Online; accessed 24 February 2019]. 132
- Wikipedia contributors. 2019a. 1973 oil crisis—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=1973_oil_crisis. [Online; accessed 25 February 2019]. 140
- Wikipedia contributors. 2019b. DEC Alpha—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DEC_Alpha. [Online; accessed 25 February 2019]. 152
- Wikipedia contributors. 2019c. Illiac—Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=ILLIAC>. [Online; accessed 25 February 2019]. 135
- Wikipedia contributors. 2019d. TeX—Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=TeX>. [Online; accessed 25 February 2019]. 144, 145
- Y. Yu. August 2018. Personal communication (recorded conversation). 153, 166
- M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pp. 75–87. ACM. DOI: [10.1145/2814576.2814733](https://doi.org/10.1145/2814576.2814733). 53
- L. Zhou. 2 August 2018. Personal communication (email). 165
- G. Ziegler. January 2000. How (La)TeX changed the face of mathematics: An e-interview with Leslie Lamport, the author of L^AT_EX. *Mitteilungen der Deutschen Mathematiker-Vereinigung*, pp. 49–51. Personal communication (email), 2 August 2018. 145

Index

Footnotes are indicated by an 'n.'

- Abadi, Martin, 10, 114
- Abortion safe behavior, 254
- Acceptors, 96
- Active replication, 85
- Admissible executions, 31–32, 34
- Agreement. *See also* Consensus
 - problem, 7–8, 68–81
 - property, 75, 86
 - protocols, 161–162
- Airplane flight control system, 7, 67, 140
- Almost-Asynchrony benign system model, 87
- Alpha architecture, 151–152
- AmazonWeb Services, 128
- Anomalous behavior, 189–191
- Antoniadis, Karolos, 343
- APL language, 135n
- Arbiter problem, 142, 243
- Arbiters, 20–22
- Armstrong, Susan, 170
- Array processing computer, 135
- Assembly language programming, 133
- Asynchronous consensus protocols, 89–96
- Asynchronous systems, 54, 59, 69
- Asynchrony benign system model, 87
- Atomicity, 4–5, 29–30, 38, 120–121
- Attiya, Hagit, 343–344
- Auxiliary variables, 113
- Averaging time values, 77
- Avionics control system, 7, 67, 140
- Bakery algorithm, 1–5, 15–17, 29–36, 136–137
- Ballots, 96, 282–287
- Basic protocol, 282, 290–291, 307–311
- Batson, Brannon, 160
- Behaviors, 107
- Ben-Or consensus protocol, 90–93
- Boolean function, 265
- Branching-time temporal logics, 106
- Brandeis University, 134
- Bravo system, 143
- Breadth-first searches, 124
- Broadcast primitives, 69
- Broadcasts, 55, 69–74, 88–89, 92, 308
- Bronx High School of Science, 131
- Bureaucrats, 298–300
- Bush, Vannevar, 131
- Bussan, Mimi, 170
- Byzantine agreement, 7–8, 68–81
- Byzantine consensus, 70
- Byzantine generals problem, 67–74, 203
- C-Agreement property, 89
- C-Completion property, 88–89
- C-Validity property, 89

- Cachin, Christian, 40, 344
 CAP theorem, 50
 Castro-Liskov algorithm, 162
 Causality, 49–52
 Chain replication, 84
 Chandy, Mani, 6, 143, 164–165
 Cheap Paxos protocol, 101, 161
 CISC (Complex Instruction Set Computers), 151–152
 Clocks, 5–6, 51, 76–77, 182–185, 189–192
 Closed systems, 114
 Coalescing transformations, 126
 Coherent cache behavior, 5
 Commanders, 69
 Commit protocols, 307
 Communication links and paths, 54–55, 217–221
 Communication variables, 240
 Compaq Computer Company, 155
 COMPASS years (1970–1977), 134–139
 Completion property, 86
 Complex Instruction Set Computers (CISC), 151–152
 Composition, 113–114
 Computer-based document production systems, 143–144
 Computer-building project, 132
 Computer Science Lab (CSL), 140
 Concurrency, 13–25, 41–43, 134, 143, 150, 156, 230, 232
 Consensus, 71. *See also* Agreement protocols, 88–89
 Consistency, 39–40, 149
 Consistent cuts, 24
 Consolidated Edison, 133
 ConstantCheck procedure, 56–57
 Constants, 107
 Control flow, 120
 Correct processes, 68–70, 86
 Counters, 137
 Crash failures, 87–88
 Critical sections, 3, 14, 16, 20–21, 30–36, 174–177, 198–199, 235, 240, 249–259
 Cryptography, 77–79
 CSL (Computer Science Lab), 140
 Cycles, 265
 Deadlock Freedom property, 251, 263, 271
 Deadlocks, 30–32, 64, 250–251, 268
 DEC/Compaq years (1985–2001), 146–155
 Decree-ordering property, 297
 Diffie, W., 77–78, 140–141
 Digital signatures, 71–74, 77–81
 Dijkstra, Edsger, 2–4, 14–15, 18, 24–25, 143, 159
 Dining philosophers problem, 3
 Dishonest legislators, 302–303
 Disk Paxos protocol, 100, 161
 Distributed algorithms, introduction, 23–24
 Distributed global state, 58–64
 Distributed snapshots, 58–63, 143
 Distributed state machine abstraction, 53–58
 Dwork, Cynthia, 94
 Dwork, Lynch, and Stockmeyer consensus protocols, 94–96
 Dynamic reconfiguration, 100–102
 Early employment years, 133–134
 Echo Distributed File System, 148
 Eclipse platform, 160
 Edsger W. Dijkstra Prize in Distributed Computing, 8, 141
 Education, 133–134
 Effectively nonconcurrent operations, 238
 Effectively precedes operations, 237
 Either primitive, 120
 Election prediction by computer, 132
 Elementary operation executions, 234
 Elements of cycles, 265
 Ellis, Clarence, 84
 Engberg, Urban, 151
 EV6 multiprocessor, 152
 EV7 multiprocessor, 153–154
 EV8 multiprocessor, 154
 Events, 19–20, 23–24, 47–52, 229–231
 Eventually consistent databases, 83

- Existential forgery, 78
- Exit conditions, 32, 250–251, 253
- Exponential information gathering, 76
- Fail-safe property, 256
- Fail-stop failures, 88
- Failure detection oracles, 88
- Failure operation, 256
- Fair keywords, 18
- Fairness requirements, 251–253
- Farsite project, 128
- Fast Paxos protocol, 161
- Fast reads, 304–306
- Fault tolerance, 24–25, 58, 67–68, 137, 140
- FCFS (first-come-first-served) condition, 252–253, 258, 269–273
- FIFO communication links, 54
- FIFO queues, 17–22, 44, 117–119, 122, 125
- Firefly project, 147
- First-come-first-served (FCFS) condition, 252–253, 258, 269–273
- Fischer, Michael, 253
- Flexible Paxos protocol, 162n
- Floyd-Hoare style of reasoning, 128
- Floyd’s method, 105
- FLP result, 148n
- Forged digital signatures, 79
- Forget actions, 312
- Formal specification and verification, 9–10, 103–105
 - impact, 127–129
 - PlusCal language, 119–122
 - TLA, 105–114
 - TLA+ language, 115–119
 - TLAPS system, 125–127
 - TLC, 123–125
- FORTRAN compilers, 134–136
- Foxboro Computers, 136n
- Frangipani system, 149
- Franken consensus, 94–95
- Gafni, Eli, 161
- Garbage collection paper, 143
- Gharachorloo, Kourosh, 152
- Gilkerson, Ellen, 170
- Glitch, The, 142
- Global state, 47–49, 58–64
- Goldberg, Jack, 141
- Google File System, 101
- Graduate education, 134
- Grain of atomicity, 120
- Grant service operations, 239
- Gray, Jim, 141, 162
- Guerraoui, Rachid, 344
- Guibas, Leo, 148
- Guindon, Dick, 162
- Happened before relations, 5, 181
- Hellman, M. E., 77–78, 140–141
- Hexagon argument, 8
- Hiding internal state, 111–113
- Hierarchical decomposition, 154
- Higher-level operation, 42
- Higher-level views, 233–234
- History variables, 10, 113
- Honest mistakes, 302–303
- Honors, 163–165
- IBM 705 computer, 133
- ILLIAC IV computer, 135–136
- Impossibility results, 72, 206–209
- Information hiding, 112
- Initial declaration, 250
- Initial states, 86
- Instantiation, 118–119
- Interactive consistency, 69–71, 73
- Interprocess communication, 234–239
- Interrupts, 136n
- Invariant predicates, 19
- IronFleet project, 128
- Jean-Claude Laprie Award in Dependable Computing, 8
- Johnson, James, 158
- Johnson, Paul R., 50, 53, 137–138
- Johnson, Samuel, 13
- Keidar, Idit, 344

- Key generation, 78–80
- Knuth, Donald, 144
- Kuppe, Markus, 160
- Ladkin, Peter, 148
- Lamport, Leslie, biography
 - collegial influences, 165–169
 - COMPASS years (1970–1977), 134–139
 - DEC/Compaq years (1985–2001), 146–155
 - early years, 131–132
 - education and early employment, 133–134
 - honors, 163–165
 - Microsoft years (2001–), 155–163
 - photo, 170–172
 - SRI years (1977–1985), 139–146
- Lamport signatures, 68, 79–81
- Lamport timestamps, 5, 51–52
- Lampson, Butler, 139n, 143, 149n, 162n, 167
- Langworthy, David, 158
- Latest update wins replication technique, 83
- LaTeX system, 11, 143–145
- Leaders, 56–57, 96–98
- Learning the law process, 300–302
- Lee, Ed, 149
- Legislator selection, 303–304
- Levin, Roy, 155, 344
- LICS Test-of-Time Award, 10
- Lieutenants, 69
- Linear-time temporal logics, 106
- Linearizability, 5
- Livelock freedom, 14
- Liveness properties, 14, 104–105, 124, 127
- Lockout Freedom Property, 251, 256
- Lockouts, 31–32, 268
- Logic TLA, 107–110
- Logical clocks, 5–6, 51, 182–185
- Logical time, 48–58
- Logical timestamps, 5
- Long ledgers, 298
- Lord, Alison, 132
- Lynch, Nancy, 94
- Majority decisions, 76
- Malfunction execution, 256
- Malfunctioning behavior, 255
- Malicious intelligence, 225
- Malkhi, Dahlia, 161, 168, 343
- Mann, Tim, 149
- Marked graphs, 21–22
- Marker messages, 60–62
- Marlboro College, 134
- Marzullo, Keith, 149, 159, 279
- Massa, Mike, 161
- Massachusetts Computer Associates (COMPASS), 134–139
- Mathematical results, 282–287
- Mathematics, graduate education in, 134
- May affect relations, 103
- Median time values, 77
- Melliar-Smith, Michael, 8
- Memory Barrier (MB) instructions, 16–17
- Memory consistency models, 5. *See also* Shared memory
- Merkle signature scheme, 80–81
- Merz, Stephan, 167, 344–345
- Metaphors, 2
- Microsoft years (2001–), 155–163
- MicroVAX line, 147
- Milner, R., 25
- Minimum delay of messages, 191
- Minkowski, H., 138
- Missing communication paths, 217–221
- MIT, 133
- MITRE Corporation, 134
- Model checkers, 123–127
- Modules, 116–119
- Molnar, Charles, 142
- Monotonicity condition, 300–302
- Moore, J., 152
- Multi-Paxos protocol, 9, 99
- Multidecree parliament, 294–297
- Multiple-reader variables, 242–243
- MultiTitan multiprocessor, 147
- Murphi model checker, 123
- Mutual exclusion
 - Bakery algorithm, 30–32

- MultiTitan multiprocessor, 147
- problem, 14–15
- vs. producer-consumer synchronization, 20–21
- proof, 16–17
- real solution, 15–16
- shared memory, 29–30
- simplification of, 136
- work on, 150
- Mutual Exclusion Problem: Statement and Solutions
 - abstract, 247
 - basic requirements, 249–251
 - conclusion, 273–275
 - failure, 254–258
 - fairness requirements, 251–253, 256
 - FCFS solutions, 269–273
 - introduction, 248
 - One-Bit Algorithm, 260–264
 - premature termination, 253–254
 - problem, 249–258
 - solutions, 258–273
 - Three-Bit Algorithm, 266–269
- Mutual Exclusion Problem: Theory of Interprocess Communication
 - abstract, 227–228
 - assumptions, 243–244
 - conclusion, 244
 - higher-level views, 233–234
 - interprocess communication, 234–239
 - introduction, 228
 - model, 228–234
 - multiple-reader variables, 242–243
 - physical considerations, 229–231
 - processes, 239–241
 - system executions, 231–233
- Mutual exclusion property, 249, 268
- Mutual exclusion protocol, 259–260
- N*-Bit Algorithm, 270–273
- N*-process systems, 21
- National Academy of Engineering honor, 164
- National Institute of Standards and Technology (NIST), 81
- Neighbors, 217
- New-old inversion behavior, 37
- Newcombe, Chris, 128, 160
- Next-time operator, 108
- NIST (National Institute of Standards and Technology), 81
- Nodes, 48
- Nonconcurrent operations, 238
- Noncritical sections, 174, 235, 240, 256–257
- Normal operation execution, 256
- Oki, Brian, 149
- One-Bit Algorithm, 260–264
- One-time signatures, 68, 80–81
- One-way functions, 79
- Open systems, 114
- Operation executions, 16–17, 22–23, 41–42, 231–233
- Operators, 116–118
- Ordered cycles, 265
- Ordering events totally, 185–189
- Ordering of decrees, 296–297
- Owicki, Susan S., 106
- p*-regular graphs, 217
- Palais, Richard, 142, 171
- Parables, 2
- Parallel composition, 113
- Parallel computing. *See* Concurrency
- Parallelism, 135–136
- Partial correctness property, 10
- Partial event ordering, 180–182
- Passed decrees, 296
- Passive replication, 85
- Password authentication, 68
- Paxos algorithm, 9, 148–150, 161–162, 164
- Paxos protocol, 96–100
- PC algorithm, 18–19
- Pease, Marshall, 7–8, 140–141
- Perfect communication links, 54
- Petal system, 149
- Petri nets, 22

- Ph.D. dissertation, 134
- Physical clocks, 190–192
- Physical considerations, 229–231
- PlusCal language, 18, 119–122, 124, 159
- Pnueli’s temporal logic, 105–106
- Post-quantum cryptosystems, 81
- Practical problems, 2
- Precedes relations, 103
- Preceding events, 230
- Premature termination, 253–254
- Presidential selection requirement, 293–294, 297–298
- Principles of Distributed Computing Influential Paper Award, 6
- Private variables, 240
- Process fairness, 14
- Process templates, 120
- Processes
 - distributed systems, 48–49
 - mutual exclusion problem, 239–241
- Producer-consumer synchronization, 17–23, 121–122
- Progress condition, 280
- Prophecy variables, 10, 113
- Proposed decrees, 296
- Public-key cryptography, 77–78
- Quantum-safe cryptosystems, 81
- Quorums, 282
 - r*-bounded waiting, 253
- Random malfunctions, 225
- Rashid, Rick, 155
- Read command, 305–306
- Read executions, 232–233
- Read-only commands, 99
- Reads, 304–306
- Real-world problems, 2
- Receive actions, 312, 314–315
- Reduced Instruction Set Computers (RISC), 151–152
- Refinement concept, 104–105, 111–112
- Registers, 4–5, 35–43
- Regular registers, 35
- Reid, Brian, 143
- Relativity, 138, 229
- Release commands, 188
- Reliable systems, 69, 221–225
- Replicated state machines, introduction, 86
- Request queues, 187–188
- Request resources, 187–188
- Request service operations, 239
- RFC 677 database, 83–84
- RISC (Reduced Instruction Set Computers), 151–152
- Rodeheffer, Tom, 158–159, 167
- Rounds, 91–95, 97–98
- RSA cryptosystem, 80–81
- Safe registers, 35, 43
- Safety properties, 10, 14, 104–105, 127
- Saxe, Jim, 152
- Schneider, Fred, 86, 139, 166–167, 170
- Schroeder, Mike, 149, 155
- Scribe document production systems, 143–145
- Secure digital signatures, 79
- Self-stabilizing algorithms, 258, 263
- Self-stabilizing multidecree parliament, 303
- Send actions, 312, 315
- Sender processes, 69
- Sequential consistency, 5, 30, 44–46, 143, 197–198
- Shared memory, 5, 29–46, 147
- Shared registers, 4–5
- Shared variables, 35–36
- Shopkeeper math, 131
- Shostak, Robert, 7–8, 141
- Shutdown behavior, 254
- Shutdown safe algorithms, 254
- SIFT (Software Implemented Fault Tolerance) project, 67, 140
- Sign operation, 78–80
- Signed messages, 213–216
- SIGOPS Hall of Fame Award, 162
- Simonyi, Charles, 143
- Single-decree Synod, 281–294, 307–311
- Slow reads, 304–306

- SMR (state machine replication) algorithm, 6, 54–58
- SMTsolvers, 126
- SMV model checker, 123
- Software Implemented Fault Tolerance (SIFT) project, 67, 140
- Software specification, 135
- Space-like planes, 230
- Space-time diagrams, 181–183
- Special relativity, 229
- SPHINCS scheme, 81
- Spin model checker, 123
- SRI years (1977–1985), 139–146
- Stable property, 64
- Standard model, 15
- Starvation, 31–32
- State machine replication (SMR) algorithm, 54–58
- State machine replication (SMR) paradigm, 6
- State machine replication with benign failures, 8–9, 83–84
 - active vs. passive replication, 85
 - asynchronous consensus protocols, 90–96
 - benign system models, 87–88
 - dynamic reconfiguration, 100–102
 - Paxos protocol, 96–100
 - protocol basics, 88–90
 - review, 85–87
- State machines, 48, 304–307
- Stockmeyer, Larry, 94
- Stoppable Paxos protocol, 102
- Strong Clock Condition, 189–191
- Strong fairness, 106
- Sturgis, Howard, 162n
- Stuttering invariance, 115, 127–129
- Stuttering steps, 105
- Synchronization, 17–23, 69, 76–77, 87, 120–122, 188, 190
- Synchronous consensus protocols, 89
- System executions, 41–42, 231–233
- System steps, 257
- Systems, concurrency, 42
- Systems Research Center (SRC), 146
- T-Agreement property, 88
- T-Completion property, 88
- T-Validity property, 88
- Taylor, Bob, 146, 155
- Temporal logic of actions (TLA), 10, 105–114, 150–151
- Terminating executions, 233
- TeX system, 144
- TeX82 system, 144
- Thekkath, Chandu, 149
- Thomas, Robert, 50, 53, 137–138
- Three-Bit Algorithm, 266–269
- Ticket numbers, 15–16, 32–35
- Time in distributed systems, 47–49
- Timestamps, 5, 50–52, 54–57, 61, 137–138
- TLA (temporal logic of actions), 10, 105–114, 150–151
 - TLA+ language, 115–127, 160–161
 - TLA+ specification, 152–153, 156–158
 - TLA+ Toolbox, 127, 159–160
 - TLAPS system, 116, 125–127, 159
 - TLC model checker, 123–125, 153–154, 156, 159–160
 - TLP tool, 151
- Total correctness property, 10
- Total delay of messages, 191
- Total order of events, 51
- Total ordering protocols, 88
- Transient malfunctions, 256
- Transition functions, 107–108
- Trying statement, 250–252, 254, 257, 268–269, 271
- Turing Award, 164
- Turing machines, 48
- Two-arrow model, 16–17, 22–23
- Two generals paradox, 141
- Two-phase commit case, 162
- Typed languages, 115–116
- Unannounced death behavior, 255
- University of Illinois array processing computer, 135

- Unobstructed exit conditions, 32
Unpredictable delay of messages, 191
Update command, 305–306
- van Renesse, Robbert, 345
Vector clocks, 52–53
Verify operation, 78, 80
Version vectors, 53
Vertical Paxos protocol, 101, 161
Viewstamped replication (VR) protocol, 99–100
Vogt, Friedrich “Fritz,” 158
- Wait-freedom condition, 4, 36–37
Weak fairness, 106
Weakening shared variables, 35–36
Web 2.0, 157
Web Services Atomic Transaction protocol, 128
- Welch, Jennifer L., 345
Western Research Lab, 147
Whorfian syndrome, 159n
Wildfire multiprocessor system, 152
World line of point objects, 229
Writers, 40–41
- XBox 360 memory system, 128
Xerox PARC, 139–140
XMSS scheme, 81
- Yu, Yuan, 152–153, 166
- Zambrovski, Simon, 160
Zenon tableau prover, 126
Zermelo-Fraenkel set theory with choice (ZFC), 115
Zhou, Lidong, 161, 165

Biographies

Dahlia Malkhi



Dahlia Malkhi received her Ph.D., an M.Sc. and a B.Sc. in computer science from the Hebrew University of Jerusalem. She is currently a lead researcher at Calibra. She is an ACM fellow (2011), serves on the Simons Institute Advisory Board, on the MIT Cryptocurrency Journal Advisory Board, and on the editorial boards of the Distributed Computing Journal. Her research career spans between industrial research and academia: 2014–2019, founding member and principal researcher at VMware Research; 2004–2014, principal researcher at Microsoft Research, Silicon Valley;

1999–2007, associate professor at the Hebrew University of Jerusalem; 1995–1999, senior researcher, AT&T Labs-Research, New Jersey. She has research interest in applied and foundational aspects of reliability and security in distributed systems.

Authors

Karolos Antoniadis is a Ph.D. candidate at EPFL under the supervision of Prof. Rachid Guerraoui. He holds an MSc in Computer Science from ETH Zurich.

Hagit Attiya received the B.Sc. degree in Mathematics and Computer Science from the Hebrew University of Jerusalem, in 1981, the M.Sc. and Ph.D. degrees in Computer Science from the Hebrew University of Jerusalem, in 1983 and 1987, respectively. She is a professor at the department of Computer Science at the Technion, Israel Institute of Technology, and holds the Harry W. Labov and Charlotte Ullman Labov Academic Chair. Before joining the Technion, she has been a post-doctoral

research associate at the Laboratory for Computer Science at MIT. Her research interests are in distributed and concurrent computing. She is the editor-in-chief of Springer's journal *Distributed Computing* and a fellow of the ACM.

Christian Cachin is a professor of computer science at the University of Bern, where he leads the cryptology and data security research group since 2019. Before that he worked for IBM Research—Zurich during more than 20 years. With a background in cryptography, he is interested in all aspects of security in distributed systems, particularly in cryptographic protocols, consistency, consensus, blockchains, and cloud-computing security. He contributed to several IBM products, formulated security standards, and helped to create the Hyperledger Fabric blockchain platform.

Rachid Guerraoui received his Ph.D. from University of Orsay, and M.Sc from University of Sorbonne, both in Computer Science. He is currently professor in Computer Science at EPFL, Switzerland. His research interests are in the area of distributed computing.

Idit Keidar received her BSc (summa cum laude), MSc (summa cum laude), and Ph.D. from the Hebrew University of Jerusalem in 1992, 1994, and 1998, respectively. She was a Postdoctoral Fellow at MIT's Laboratory for Computer Science. She is currently a Professor at the Technion's Viterbi Faculty of Electrical Engineering, where she holds the Lord Leonard Wolfson Academic Chair. She serves as the Head of the Technion Rothschild Scholars Program for Excellence, and also heads the EE Faculty's EMET Excellence Program. Her research interests are in fault-tolerant distributed and concurrent algorithms and systems, theory and practice. Recently, she is mostly interested in distributed storage and concurrent data structures and transactions.

Roy Levin is a retired Distinguished Engineer, Managing Director, and founder of Microsoft Research Silicon Valley (2001–2014). Previously (1984–2001) he was a senior researcher and later Director of the Digital/Compaq Systems Research Center in Palo Alto, California. Before joining Digital, Levin was a researcher at Xerox's Palo Alto Research Center (1977–1984). His research focus was distributed computing systems. Levin holds a Ph.D. in Computer Science from Carnegie-Mellon University and a B.S. in Mathematics from Yale University. He is a Fellow of the Association for Computing Machinery (ACM) and a former chair of Special Interest Group on Operating Systems (SIGOPS).

Stephan Merz received his Ph.D. and habilitation in computer science from the University of Munich, Germany. He is currently a senior researcher and the head of science at Inria Nancy–Grand Est, France. His research interests are in formal veri-

fication of distributed algorithms and systems using model checking and theorem proving.

Robbert van Renesse received his Ph.D. in computer science, and an M.Sc. and a B.Sc. in mathematics from the Vrije Universiteit Amsterdam. He is currently a Research Professor at the Department of Computer Science at Cornell University and is Associate Editor of ACM Computing Surveys. He is an ACM fellow (since 2009). He was a researcher at AT&T Bell Labs in Murray Hill (1990) and served as Chair of ACM SIGOPS (2015–2019). His research interests include reliable distributed systems and operating systems.

Jennifer L. Welch received her S.M. and Ph.D. from the Massachusetts Institute of Technology and her B.A. from the University of Texas at Austin. She is currently holder of the Chevron II Professorship and Regents Professorship in the Department of Computer Science and Engineering at Texas A&M University, and is an ACM Distinguished Member. Her research interests are in the theory of distributed computing, algorithm analysis, distributed systems, mobile ad hoc networks, and distributed data structures.

Concurrency

The Works of Leslie Lamport

Dahlia Malki, Editor

This book is a celebration of Leslie Lamport's work on concurrency, interwoven in four-and-a-half decades of an evolving industry: from the introduction of the first personal computer to an era when parallel and distributed multiprocessors are abundant. His works lay formal foundations for concurrent computations executed by interconnected computers. Some of the algorithms have become standard engineering practice for fault tolerant distributed computing – distributed systems that continue to function correctly despite failures of individual components. He also developed a substantial body of work on the formal specification and verification of concurrent systems, and has contributed to the development of automated tools applying these methods.

Part I consists of technical chapters of the book and a biography. The technical chapters of this book present a retrospective on Lamport's original ideas from experts in the field. Through this lens, it portrays their long-lasting impact. The chapters cover timeless notions Lamport introduced: the Bakery algorithm, atomic shared registers and sequential consistency; causality and logical time; Byzantine Agreement; state machine replication and Paxos; temporal logic of actions (TLA). The professional biography tells of Lamport's career, providing the context in which his work arose and broke new grounds, and discusses LaTeX – perhaps Lamport's most influential contribution outside the field of concurrency. This chapter gives a voice to the people behind the achievements, notably Lamport himself, and additionally the colleagues around him, who inspired, collaborated, and helped him drive worldwide impact. Part II consists of a selection of Leslie Lamport's most influential papers.

This book touches on a lifetime of contributions by Leslie Lamport to the field of concurrency and on the extensive influence he had on people working in the field. It will be of value to historians of science, and to researchers and students who work in the area of concurrency and who are interested to read about the work of one of the most influential researchers in this field.

ABOUT ACM BOOKS



ACM Books is a series of high-quality books published by ACM for the computer science community. ACM Books publications are widely distributed in print and digital formats by major booksellers and are available to libraries and library consortia. Individual ACM members may access ACM Books publications via separate annual subscription.

ISBN 978-1-4503-7271-8

90000



9 781450 372718

Constructing Digital Signatures from a One Way Function

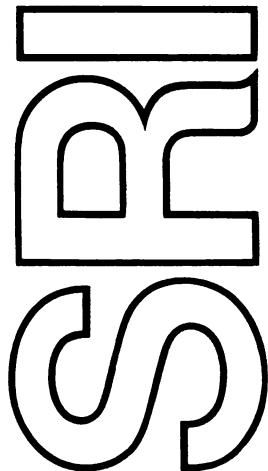
Leslie Lamport

Computer Science Laboratory

SRI International

18 October 1979

CSL - 98



333 Ravenswood Ave. • Menlo Park, California 94025
(415) 326-6200 • Cable: SRI INTL MPK • TWX: 910-373-1246

1. Introduction

A digital signature created by a sender P for a document m is a data item $\sigma_P(m)$ having the property that upon receiving m and $\sigma_P(m)$, one can determine (and if necessary prove in a court of law) that P generated the document m .

A one way function is a function that is easy to compute, but whose inverse is difficult to compute [1]. More precisely a one way function ϕ is a function from a set of data objects to a set of values having the following two properties:

1. Given any value v , it is computationally infeasible to find a data object d such that $\phi(d) = v$.
2. Given any data object d , it is computationally infeasible to find a different data object d' such that $\phi(d') = \phi(d)$.

If the set of data objects is larger than the set of values, then such a function is sometimes called a one way hashing function.

We will describe a method for constructing digital signatures from such a one way function ϕ . Our method is an improvement of a method devised by Rabin [2]. Like Rabin's, it requires the sender P to deposit a piece of data α in some trusted public repository for each document he wishes to sign. This repository must have the following properties:

- α can be read by anyone who wants to verify P 's signature.
- It can be proven in a court of law that P was the creator of α .

Once α has been placed in the repository, P can use it to generate a signature for any single document he wishes to send.

Rabin's method has the following drawbacks not present in ours.

1. The document m must be sent to a single recipient Q , who then requests additional information from P to validate the signature. P cannot divulge any additional validating information without compromising information that must remain private to prevent someone else from generating a new document m' with a valid signature $\sigma_P(m')$.
2. For a court of law to determine if the signature is valid, it is necessary for P to give the court additional private information.

This has the following implications.

- P -- or a trusted representative of P -- must be available to the court.
- P must maintain private information whose accidental disclosure would enable someone else to forge his signature on a document.

With our method, P generates a signature that is verifiable by anyone, with no further action on P's part. After generating the signature, P can destroy the private information that would enable someone else to forge his signature. The advantages of our method over Rabin's are illustrated by the following considerations when the signed document m is a check from P payable to Q .

1. It is easy for Q to endorse the check payable to a third party R by sending him the signed message "make m payable to R ". However, with Rabin's scheme, R cannot determine if the check m was really signed by P , so he must worry about forgery by Q as well as whether or not P can cover the check. With our method, there is no way for Q to forge the check, so the endorsed check is as good as a check payable directly to R signed by P . (However, some additional mechanism must be introduced to prevent Q from cashing the original check after he has signed it over to R .)
2. If P dies without leaving the executors of his estate the information he used to generate his signatures, then Rabin's method cannot prevent Q from undetectably altering the check m -- for example, by changing the amount of money payable. Such posthumous forgery is impossible with our method.
3. With Rabin's method, to be able to successfully challenge any attempt by Q to modify the check before cashing it, P must maintain the private information he used to generate his signature. If anyone (not just Q) stole that information, that person could forge a check from P payable to him. Our method allows P to destroy this private information after signing the check.

2. The Algorithm

We assume a set M of possible documents, a set K of possible keys¹,

¹The elements of K are not keys in the usual cryptographic sense, but are arbitrary data items. We call them keys because they play the same role as the keys in Rabin's algorithm.

and a set \underline{V} of possible values. Let Σ denote the set of all subsets of $\{1, \dots, 40\}$ containing exactly 20 elements. (The numbers 40 and 20 are arbitrary, and could be replaced by $2n$ and n . We are using these numbers because they were used by Rabin, and we wish to make it easy for the reader to compare our method with his.)

We assume the following two functions.

1. A function $F : \underline{K} \rightarrow \underline{V}$ with the following two properties:

- a. Given any value v in \underline{V} , it is computationally infeasible to find a key k in \underline{K} such that $F(k) = v$.
- b. For any small set of values v_1, \dots, v_m , it is easy to find a key k such that $F(k)$ is not equal to any of the v_i .

2. A function $G : \underline{M} \rightarrow \Sigma$ with the property that given any document m in \underline{M} , it is computationally infeasible to find a document $m' \neq m$ such that $G(m') = G(m)$.

For the function F , we can use any one way function ϕ whose domain is the set of keys. The second property of F follows easily from the second property of the one way function ϕ . We will discuss later how the function G can be constructed from an ordinary one way function.

For convenience, we assume that P wants to generate only a single signed document. Later, we indicate how he can sign many different documents. The sender P first chooses 40 keys k_i such that all the values $F(k_i)$ are distinct. (Our second assumption about F makes this easy to do.) He puts in a public repository the data item $\alpha = (F(k_1), \dots, F(k_{40}))$. Note that P does not divulge the keys k_i , which by our first assumption about F cannot be computed from α .

To generate a signature for a document m , P first computes $G(m)$ to obtain a set $\{i_1, \dots, i_{20}\}$ of integers. The signature consists of the 20 keys $k_{i_1}, \dots, k_{i_{20}}$. More precisely, we have $\sigma_P(m) = (k_{i_1}, \dots, k_{i_{20}})$, where the i_j are defined by the following two requirements:

- (i) $G(m) = \{i_1, \dots, i_{20}\}$.

(ii) $i_1 < \dots < i_{20}$

After generating the signature, P can destroy all record of the 20 keys k_s with s not in $G(m)$.

To verify that a 20-tuple (h_1, \dots, h_{20}) is a valid signature $\sigma_p(m)$ for the document m , one first computes $G(m)$ to find the i_j and then uses α to check that for all j , h_j is the i_j th key. More precisely, the signature is valid if and only if for each j with $1 \leq j \leq 20$:

$F(h_j) = \alpha_{i_j}$, where α_i denotes the i th component of α , and the i_j are defined by the above two requirements.

To demonstrate that this method correctly implements digital signatures, we prove that it has the following properties.

1. If P does not reveal any of the keys k_i , then no-one else can generate a valid signature $\sigma_p(m)$ for any document m .
2. If P does not reveal any of the keys k_j except the ones that are contained in the signature $\sigma_p(m)$, then no-one else can generate a valid signature $\sigma_p(m')$ for any document $m' \neq m$.

The first property is obvious, since the signature $\sigma_p(m)$ must contain 20 keys k_i such that $F(k_i) = \alpha_i$, and our first assumption about F states that it is computationally infeasible to find the keys k_i just knowing the values $F(k_i)$.

To prove the second property, note that since no-one else can obtain any of the keys k_i , we must have $\sigma_p(m') = \sigma_p(m)$. Moreover, since the α_i are all distinct, for the validation test to be passed by $\sigma_p(m')$, we must also have $G(m') = G(m)$. However, our assumption about G states that it is computationally infeasible to find such a document m' . This proves the correctness of our method.

For P to send many different documents, he must use a different α for each one. This means that there must be a sequence of 40-tuples $\alpha_1, \alpha_2, \dots$ and the document must indicate which α_i is used to generate that document's

signature. The details are simple, and will be omitted.

3. Constructing the Function G

One way functions have been proposed whose domain is the set of documents and whose range is a set of integers of the form $\{0, \dots, 2^n-1\}$, for any reasonably large value of n . (It is necessary for n to be large enough to make exhaustive searching over the range of ϕ computationally infeasible.) Such functions are described in [1] and [2]. The obvious way to construct the required function G is to let ϕ be such a one way function, and define $G(m)$ to equal $R(\phi(m))$, where $R : \{0, \dots, 2^n-1\} \rightarrow \Sigma$.

It is easy to construct a function R having the required range and domain. For example, one can compute $R(s)$ inductively as follows:

1. Divide s by 40 to obtain a quotient q and a remainder r
2. Use r to choose an element x from $\{1, \dots, 40\}$. (This is easy to do, since $0 \leq r \leq 40$.)
3. Use q to choose 19 elements from the set $\{1, \dots, 40\} - \{x\}$ as follows:
 - a. Divide q by 39 to obtain a quotient ...

It requires a careful analysis of the properties of the one way function ϕ to be sure that the resulting function G has the required property. We suspect that for most one way functions ϕ , this method would work. However, we cannot prove this.

The reason constructing G in this manner might not work is that the function R from $\{0, \dots, 2^n\}$ into Σ is a many to one mapping, and the resulting "collapsing" of the domain might defeat the one way nature of ϕ . However, it is easy to show that if the function R is one to one, then property (ii) of ϕ implies that G has the required property. To construct G , we need only find an easily computable one to one function R from $\{0, \dots, 2^n-1\}$ into Σ , for a reasonably large value of n .

We can simplify our task by observing that the function G need not be defined on the entire set of documents. It suffices that for any document

m , it is easy to modify m in a harmless way to get a new document that is in the domain of G . For example, one might include a meaningless number as part of the document, and choose different values of that number until he obtains a document that is in the domain of G . This is an acceptable procedure if (i) it is easy to determine whether a document is in the domain, and (ii) the expected number of choices one must make before finding a document in the domain is small.

With this in mind, we let $n = 40$ and define $R(s)$ as follows: if the binary representation of s contains exactly 20 ones, then $R(s) = \{i : \text{the } i\text{th bit of } s \text{ equals one}\}$, otherwise $R(s)$ is undefined. Approximately 13% of all 40 bit numbers contain exactly 20 ones. Hence, if the one way function ϕ is sufficiently randomizing, there is a .13 probability that any given document will be in the domain of G . This means that randomly choosing documents (or modifications to a document), the expected number of choices before finding one in the domain of G is approximately 8. Moreover, after $17p$ choices, the probability of not having found a document in the domain of G is about $1/10^p$. (If we use 60 keys instead of 40, the expected number of choices to find a document in the domain becomes about 10, and $22p$ choices are needed to reduce the probability of not finding one to $1/10^p$.)

If the one way function ϕ is easy to compute, then these numbers indicate that the expected amount of effort to compute G is reasonable. However, it does seem undesirable to have to try so many documents before finding one in the domain of G . We hope that someone can find a more elegant method for constructing the function G , perhaps by finding a one to one function R which is defined on a larger subset of $\{0, \dots, 2^n\}$.

Note: We have thus far insisted that $G(m)$ be a subset of $\{1, \dots, 40\}$ consisting of exactly 20 elements. It is clear that the generation and verification procedure can be applied if $G(m)$ is any proper subset. An examination of our correctness proof shows that if we allow $G(m)$

to have any number of elements less than 40, then our method would still have the same correctness properties if G satisfies the following property:

- For any document m , it is computationally infeasible to find a different document m' such that $G(m')$ is a subset of $G(m)$.

By taking the range of G to be the collection of 20 element subsets, we insure that $G(m')$ cannot be a proper subset of $G(m)$. However, it may be possible to construct a function G satisfying this requirement without constraining the range of G in this way.

REFERENCES

- [1] Diffie, W. and Hellman, M. "New Directions in Cryptography". IEEE Trans. on Information Theory IT-22 (November 1976), 644-654.
- [2] Rabin, M. "Digitalized Signatures", in Foundations of Secure Computing, Academic Press (1978), 155-168.

REFERENCES

- [1] C. Black, C.-E. Sundberg, and W. K. S. Walker, "Development of a spaceborne memory with a single error and erasure correction scheme," in *Conf. Rec., 1977 Fault-Tolerant Computing Symp.*, FTCS-7, June 1977, pp. 50-55.
- [2] C. Black *et al.*, "Highly reliable semiconductor memory for the on-board computer," Final Rep. ESTEC Contr. 2528/75/HP, British Aircraft Corp., Bristol, England.
- [3] R. W. Lucky, J. Salz, and E. J. Weldon, Jr., *Principles of Data Communication*, New York: McGraw-Hill, 1968.
- [4] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd Ed. Cambridge, MA: MIT Press, 1972.
- [5] M. Y. Hsiao, "A class of optimal minimum odd weight column SEC/DED codes," *IBM J. Res. Develop.*, pp. 395-401, July 1970.
- [6] G. Longo, "Algebraic coding and combinatorics: A tutorial survey," NATO Advanced Study Institute, Darlington, England, Aug. 8-20, 1977, *Conf. Rec.*, Sijhoff & Noordhoff, 1978, series E, no. 25, pp. 151-169.
- [7] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.
- [8] R. P. Copee, "Memory-rich minis get aggressive," *Electronics*, pp. 69-70, Aug. 18, 1977.
- [9] H. J. Helgert and R. D. Stinnett, "Minimum distance bounds for binary linear codes," *IEEE Trans. Information Theory*, vol. IT-19, pp. 344-356, May 1973.
- [10] C.-E. Sundberg, "Some transparent shortened codes for semiconductor memories," *Telecommunication Theory*, Univ. of Lund, Lund, Sweden, Tech. Rep. TR-91, Sept. 1977.
- [11] —, "Erasures and errors decoding for semiconductor memories," *IEEE Trans. Comput.*, vol. C-27, pp. 696-705, Aug. 1978.
- [12] W. C. Carter and C. E. McCarthy, "Implementation of an experimental fault-tolerant memory system," *IEEE Trans. Comput.*, vol. C-25, pp. 557-568, June 1976.

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

Index Terms—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]-[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*. The sequentiality

Manuscript received September 28, 1977; revised May 8, 1979.
The author is with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

We assume that the computer consists of a collection of processors and memory modules, and that the processors communicate with one another only through the memory modules. (Any special communication registers may be regarded as separate memory modules.) The only processor operations that concern us are the operations of sending fetch and store requests to memory modules. We assume that each processor issues a sequence of such requests. (It must sometimes wait for requests to be executed, but that does not concern us.)

We illustrate the problem by considering a simple two-process mutual exclusion protocol. Each process contains a *critical section*, and the purpose of the protocol is to insure that only one process may be executing its critical section at any time. The protocol is as follows.

process 1

```
a := 1;  
if b = 0 then critical section;  
    a := 0  
else ... fi
```

process 2

```
b := 1;  
if a = 0 then critical section;  
    b := 0  
else ... fi
```

The else clauses contain some mechanism for guaranteeing eventual access to the critical section, but that is irrelevant to the discussion. It is easy to prove that this protocol guarantees mutually exclusive access to the critical sections. (Devising a proof provides a nice exercise in using the assertional techniques of [2] and [3], and is left to the reader.) Hence, when this two-process program is executed by a sequentially consistent multiprocessor computer, the two processors cannot both be executing their critical sections at the same time.

We first observe that a sequential processor could execute the "b := 1" and "fetch b" operations of process 1 in either order. (When process 1's program is considered by itself, it does not matter in which order these two operations are performed.) However, it is easy to see that executing the "fetch b" operation first can lead to an error—both processes could then execute their critical sections at the same time. This immediately suggests our first requirement for a multiprocessor computer.

Requirement R1: Each processor issues memory requests in the order specified by its program.

Satisfying Requirement R1 is complicated by the fact that storing a value is possible only after the value has been computed. A processor will often be ready to issue a memory fetch request before it knows the value to be stored by a preceding store request. To minimize waiting, the processor can issue the store request to the memory module without specifying the value to be stored. Of course, the store request cannot actually be executed by the memory module until it receives the value to be stored.

Requirement R1 is not sufficient to guarantee correct execution. To see this, suppose that each memory module has several ports, and each port services one processor (or I/O channel). Let the values of "a" and "b" be stored in separate memory modules, and consider the following sequence of events.

- 1) Processor 1 sends the "a := 1" request to its port in memory module 1. The module is currently busy executing an operation for some other processor (or I/O channel).
- 2) Processor 1 sends the "fetch b" request to its port in memory module 2. The module is free, and execution is begun.
- 3) Processor 2 sends its "b := 1" request to memory module 2. This request will be executed after processor 1's "fetch b" request is completed.
- 4) Processor 2 sends its "fetch a" request to its port in memory module 1. The module is still busy.

There are now two operations waiting to be performed by memory module 1. If processor 2's "fetch a" operation is performed first, then both processes can enter their critical sections at the same time, and the protocol fails. This could happen if the memory module uses a round robin scheduling discipline in servicing its ports.

In this situation, an error occurs only if the two requests to memory module 1 are not executed in the same order in which they were received. This suggests the following requirement.

Requirement R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Condition R1 implies that a processor may not issue any further memory requests until after its current request has been entered on the queue. Hence, it must wait if the queue is full. If two or more processors are trying to enter requests in the queue at the same time, then it does not matter in which order they are serviced.

Note. If a fetch requests the contents of a memory location for which there is already a write request on the queue, then the fetch need not be entered on the queue. It may simply return the value from the last such write request on the queue. \square

Requirements R1 and R2 insure that if the individual processors are sequential, then the entire multiprocessor computer is sequentially consistent. To demonstrate this, one first introduces a relation \rightarrow on memory requests as follows. Define $A \rightarrow B$ if and only if 1) A and B are issued by the same processor and A is issued before B, or 2) A and B are issued to the same memory module, and A is entered in the queue before B (and is thus executed before B). It is easy to see that R1 and R2 imply that \rightarrow is a partial ordering on the set of memory requests. Using the sequentiality of each processor, one can then prove the following result: each fetch and store operation fetches or stores the same value as if all the operations were executed sequentially in any order such that $A \rightarrow B$ implies that A is executed before B. This in turn proves the sequential consistency of the multiprocessor computer.

Requirement R2 states that a memory module's request queue must be serviced in a FIFO order. This implies that the memory module must remain idle if the request at the head of its queue is a store request for which the value to be stored has not yet been received. Condition R2 can be weakened to allow the memory module to service other requests in this situation. We need only require that all requests to the same memory cell be serviced in the order that they appear in the queue. Requests to different memory cells may be serviced out of order. Sequential consistency is

preserved because such a service policy is logically equivalent to considering each memory cell to be a separate memory module with its own request queue. (The fact that these modules may share some hardware affects the rate at which they service requests and the capacity of their queues, but it does not affect the logical property of sequential consistency.)

The requirements needed to guarantee sequential consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

REFERENCES

- [1] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115-138, 1971.
- [2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 125-143, Mar. 1977.
- [3] S. Owicky and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. Assoc. Comput. Mach.*, vol. 19, pp. 279-285, May 1976.

Comments on "An Approach to Highly Integrated Computer-Maintained Cellular Arrays"

VISHWANI D. AGRAWAL

Abstract—The above paper¹ describes machines constructed on faulty logic arrays. These machines use some or all of the good cells that form a connective cluster. It is pointed out here that when the faulty cells are randomly distributed over the array, the propagation of a freely expanding signal, which must avoid the faulty cells, is a percolation process. The percolation process determines the limit to the size of a machine embedded in a faulty array. Some useful characteristics of this process are discussed.

Index Terms—Cellular arrays, faults in logic arrays, percolation process, programmable logic, random processes.

Manning, in a recent paper¹, has described the embedded machines which are configured in a logic array containing some flawed cells. In a rectangular array, each cell communicates through its four neighbors and hence becomes inaccessible if all the four neighbors are flawed. The embedded machine thus uses a connective cluster of logic cells bounded within the array boundaries and the flawed cells. As the number of flawed cells increases, the size of a good cell cluster becomes restricted. The purpose of this communication is to illustrate that the formation of an embedded cluster in an array of flawed cells is a *percolation process* which has been extensively studied in the literature [1].

Manuscript received November 20, 1977; revised April 20, 1979.

The author was with TRW Defense and Space Systems Group, Redondo Beach, CA 90278. He is now with the Bell Laboratories, Murray Hill, NJ 07974.

¹ F. B. Manning, *IEEE Trans. Comput.*, vol. C-26, pp. 536-552, June 1977.

How to Write a 21st Century Proof

Leslie Lamport

23 November 2011
Minor change on 15 January 2012

Abstract

A method of writing proofs is described that makes it harder to prove things that are not true. The method, based on hierarchical structuring, is simple and practical. The author's twenty years of experience writing such proofs is discussed.

Contents

1	Introduction	1
2	An Example	2
3	Hierarchical Structure	6
4	A Language for Structured Proofs	9
4.1	The Proof Steps of TLA ⁺	9
4.2	Hierarchical Numbering	13
4.3	Equational Proofs	14
4.4	Comments	14
4.5	Completely Formal Proofs	15
5	Experience	16
6	Objections to Structured Proof	18
7	Beginning	19
	Acknowledgement	19
	References	19
	Appendix: A Formal Proof	21

In addition to developing the students' intuition about the beautiful concepts of analysis, it is surely equally important to persuade them that precision and rigor are neither deterrents to intuition, nor ends in themselves, but the natural medium in which to formulate and think about mathematical questions.

Michael Spivak, *Calculus* [7]

1 Introduction

Some 20 years ago, I published an article titled *How to Write a Proof* in a festschrift in honor of the 60th birthday of Richard Palais [5]. In celebration of his 80th birthday, I am describing here what I have learned since then about writing proofs and explaining how to write them.

As I observed in the earlier article, mathematical notation has changed considerably in the last few centuries. Mathematicians no longer write formulas as prose, but use symbolic notation such as $e^{i\pi} + 1 = 0$. On the other hand, proofs are still written in prose pretty much the way they were in the 17th century. The proofs in Newton's *Principia* seem quite modern. This has two consequences: proofs are unnecessarily hard to understand, and they encourage sloppiness that leads to errors.

Making proofs easier to understand is easy. It requires only the simple application of two principles: structure and naming. When one reads a sentence in a prose proof, it is often unclear whether the sentence is asserting a new fact or justifying a previous assertion; and if it asserts a new fact, one has to read further to see if that fact is supposed to be obviously true or is about to be proved. Adding structure makes it clear what the function of each assertion is. If an assertion follows from previously stated facts, in a prose proof it is often unclear what those facts are. Naming all facts makes it easy to tell the reader exactly which ones are being used.

Introducing structure has another important benefit. When writing a proof, we are continually deciding how detailed an explanation to provide the reader. Additional explanation helps the reader understand what is being shown at that point in the proof. However, in a prose proof, the additional length makes it harder to follow the logic of the complete proof. Proper structuring allows us to add as much detailed explanation as we like without obscuring the larger picture.

Eliminating errors from proofs is not as easy. It takes precision and rigor, which require work. Structuring proofs makes it possible to avoid errors; hard work is needed to make it probable.

One mistake I made in the earlier article was to advocate making proofs both easier to read and more rigorous. Learning both a new way to write proofs and how to be more precise and rigorous was too high a barrier for most mathematicians. I try here to separate the two goals. I hope that structuring proofs to be easier to read will make mathematicians more aware of how sloppy their proofs are and encourage greater precision and rigor. But the important goal is to stop writing 17th century prose proofs in the 21st century.

Another mistake I made was giving the impression that I know the best way of writing proofs. I claim to have a *better* way to write proofs than mathematicians now use. In the 21st century, it is not hard to improve on 17th century proofs. What I describe here is what I have learned in over 20 years of writing structured proofs. I am sure my way of writing proofs can be improved, and I encourage mathematicians to improve it. They will not do it by remaining stuck in the 17th century.

The way I now write proofs has profited by my recent experience designing and using a formal language for machine-checked proofs. Mathematicians will not write completely formal proofs in the next 20 years. However, learning how to write a formal proof can teach how to write ordinary, informal ones. Formal proofs are therefore briefly discussed in Section 4, and the appendix contains a formal, machine-checked proof of the example introduced in Section 2.

I am writing this at the end of an era. For millenia, mathematics has been recorded on the processed remains of dead plants and animals. By the end of the 21st century, mathematical books and articles will be read almost exclusively on electronic devices—or perhaps using even more advanced technology. This will enable the use of hypertext, which is the natural medium for structured proofs. However, in the next decade or two, mathematics will still be printed on paper or disseminated as static electronic images of printed pages. I therefore concentrate on how to write proofs as conventional printed text, with only brief mention of hypertext.

2 An Example

To illustrate how to structure a proof, let us convert a simple prose proof into a structured one that is easier to read. The example I have chosen is adapted from the proof of a corollary to the Mean Value Theorem from Spivak's *Calculus* [7, page 170]; it is shown in Figure 1. (The original corollary covered both the case of increasing and decreasing functions, but Spivak proved only

Corollary If $f'(x) > 0$ for all x in an interval, then f is increasing on the interval.

Proof Let a and b be two points in the interval with $a < b$. Then there is some x in (a, b) with

$$f'(x) = \frac{f(b) - f(a)}{b - a}.$$

But $f'(x) > 0$ for all x in (a, b) , so

$$\frac{f(b) - f(a)}{b - a} > 0.$$

Since $b - a > 0$ it follows that $f(b) > f(a)$. \blacksquare

Figure 1: Spivak's corollary and proof.

the increasing case; that proof is copied verbatim.) For compactness, I refer to the corollary and proof as Spivak's.

I chose this proof because it is short and simple. I would not normally structure such a simple four-sentence proof because the mathematically sophisticated readers of my papers would have no trouble understanding it. However, Spivak was writing for first-year calculus students. Moreover, seeing that even this simple proof is not so easy to understand should make it clear that structuring is needed for more complicated proofs.

We structure this proof as a sequence of named statements, each with a proof, and we adopt the conventional approach of using sequential numbers as statement names. The first numbered statement is the first sentence of the prose proof:

1. Let a and b be two points in the interval with $a < b$.

When we choose something in a proof, we have to prove that it exists. The proof of this statement should therefore prove the existence of points a and b in the interval with $a < b$. However, this can't be proved. The corollary assumes an interval, but it doesn't assume that the interval contains two different points. The book's definition of an interval allows a single point and the empty set as intervals. We can't prove that the interval contains two distinct points because it needn't.

A mathematician, examining the entire proof, will realize that we don't really have to prove that the interval contains two distinct points. However, this is not obvious to a beginning calculus student. A proof is not easy

to understand if we must read the entire proof to understand why its first sentence is justified. Let us just note this problem for now; we'll fix it later.

The second statement of the structured proof comes from the second sentence of the prose proof:

$$2. \text{ There is some } x \text{ in } (a, b) \text{ with } f'(x) = \frac{f(b) - f(a)}{b - a}.$$

Spivak gives no justification for this assertion; the “Then” that begins the sentence alerts readers that it follows from facts that preceded it. The reader must deduce that those facts are: a and b are in the interval, f is differentiable on the interval, and the Mean Value Theorem. Instead of forcing the reader to deduce them, we make the proof easier to understand by stating these facts in the following proof of the statement:

PROOF: By 1, the corollary's hypothesis, and the Mean Value Theorem.

The prose proof's third sentence, “But $f'(x) > 0 \dots$ ” asserts two facts, so let's turn them into steps 3 and 4 of the structured proof. The final sentence could also be viewed as two statements, but I find it more natural to turn it into the assertion $f(b) > f(a)$ and its proof. Adding these statements and their proofs, we obtain the simple structured proof of Figure 2.

Examine the structured version. Except for the $b - a > 0$ in the proof of step 5, all the explanation in the steps' proofs is missing from the original. Adding that explanation makes the proof easier to understand. It might seem excessive to a mathematician, who can easily fill in those missing justifications. However, a web search reveals that Spivak's text, while highly regarded for its rigor, is considered very difficult for most students. Those students would appreciate the additional explanation.

Adding the explanations missing from Spivak's proof does make the structured proof somewhat longer, occupying about 40% more vertical space in the typeset version I am now viewing. However, the extra length does not obscure the proof's structure. It is easy to ignore the subproofs and read just the five steps. I find the structure more apparent in the rewritten version than in the original, and I think others will too—especially once they get used to reading structured proofs.

The advantage of the structured version becomes much more obvious with hypertext. When books are read as hypertext, readers will first see the corollary with no proof. They will click or tap on the corollary or some adjacent icon to view the proof. With the structured version, they will then see just the five steps—essentially the same proof as Spivak's, except with

Corollary If $f'(x) > 0$ for all x in an interval, then f is increasing on the interval.

1. Let a and b be two points in the interval with $a < b$.

PROOF: ???

2. There is some x in (a, b) with $f'(x) = \frac{f(b) - f(a)}{b - a}$.

PROOF: By 1, the corollary's hypothesis, and the Mean Value Theorem.

3. $f'(x) > 0$ for all x in (a, b) .

PROOF: By the hypothesis of the corollary and 1.

4. $\frac{f(b) - f(a)}{b - a} > 0$

PROOF: By 2 and 3.

5. $f(b) > f(a)$

PROOF: By 1, which implies $b - a > 0$, and 4.

Figure 2: Adding structure to Spivak's proof.

the omission of “Since $b - a > 0$ it follows that”. They can then view and hide the proofs of individual steps as they choose. (They can also choose to view the complete proof.)

We are not through with the proof. We cannot prove step 1. This problem is a symptom of a glaring omission in the proof. Has the reader observed that neither Spivak's proof nor its structured version provides the slightest hint of why the proof actually proves the corollary? Readers must figure that out by themselves. Leaving such an important step to the reader does not make the proof easy to understand.

This gaffe is impossible in the proof style that I propose. The style requires that the last step of every proof be a statement of what it is that the proof is trying to prove—in this case, the statement of the corollary. Instead of repeating what we're trying to prove, which has already been stated, we write Q.E.D. (that which was to be shown) instead. A structured proof of the corollary cannot end as in Figure 2, with an assertion that is not what we are trying to prove.

The reader may have surmised that what is missing from Spivak's proof is an explanation of why we should choose a and b . His first sentence, and our first step, should have been:

If suffices to assume that a and b are two points in the interval with $a < b$ and prove $f(b) > f(a)$.

The justification of that step is that it follows from the definition of an increasing function. Step 5 can then be replaced by:

5. Q.E.D.

PROOF: Step 1 implies $b - a > 0$, so 4 implies $f(b) - f(a) > 0$, which implies $f(b) > f(a)$. By 1, this proves the corollary.

We can further improve the proof by making more use of naming. Let's give the nameless interval in the statement of the corollary the name I . Also, step 1 asserts two assumptions: that a and b are in I and that $a < b$. Let's name those assumptions so we can refer in the proof to the exact one that is being used. The result is the proof in Figure 3.

Transforming Spivak's proof to the structured proof in Figure 3 was quite simple. We could have done it differently—for example, by making $b - a > 0$ a separate step, or by removing a bit of the explanation and using this proof of step 5:

PROOF: Assumption 1.2 implies $b - a > 0$, so 4 implies $f(b) > f(a)$.
By 1, this proves the corollary.

With any sensible choices, the resulting structured proof would be easier to understand than Spivak's unstructured proof.

3 Hierarchical Structure

Writing the structured proof of Figure 3 forced us to write a justification for each step. Having to do this helps catch errors. However, it is not enough to ensure error-free proofs. In fact, our structured proof contains an important omission.

The best way I know to eliminate errors is to imagine that there is a curious child sitting next to us. Every time we write an assertion, the child asks: *Why?* When we wrote the proof of step 2, the child would ask “Why does step 2 follow from the assumptions and the Mean Value Theorem?” To answer, we would point to Spivak's statement of the Mean Value Theorem:¹

¹The astute reader will notice that this theorem assumes the unstated hypothesis $a < b$. When introducing the notation (a, b) , the book states that $a < b$ “is almost always assumed (explicitly if one has been careful, implicitly otherwise).”

Corollary If $f'(x) > 0$ for all x in an interval I , then f is increasing on I .

1. It suffices to assume

1. a and b are points in I
2. $a < b$

and prove $f(b) > f(a)$.

PROOF: By definition of an increasing function.

2. There is some x in (a, b) with $f'(x) = \frac{f(b) - f(a)}{b - a}$.

PROOF: By assumptions 1.1 and 1.2, the hypothesis that f is differentiable on I , and the Mean Value Theorem.

3. $f'(x) > 0$ for all x in (a, b) .

PROOF: By the hypothesis of the corollary and assumption 1.1.

4. $\frac{f(b) - f(a)}{b - a} > 0$

PROOF: By 2 and 3.

5. Q.E.D.

PROOF: Assumption 1.2 implies $b - a > 0$, so 4 implies $f(b) - f(a) > 0$, which implies $f(b) > f(a)$. By 1, this proves the corollary.

Figure 3: Fixing the proof.

Theorem 4 (The Mean Value Theorem) If f is continuous on $[a, b]$ and differentiable on (a, b) , then there is a number x in (a, b) such that

$$f'(x) = \frac{f(b) - f(a)}{b - a}$$

Step 2 is identical to the conclusion of the theorem, but the child would ask why the hypotheses hold. In particular, why is f continuous on $[a, b]$? We would answer, “Because f is differentiable on I .” “But why does that imply f is continuous?” We would turn back 36 pages in the book and point to Spivak’s Theorem 1, which asserts that differentiability implies continuity. “You didn’t tell me you needed Theorem 1.” The child is right. If that result is stated as a numbered theorem, surely its use should be mentioned. The proof of step 2 should be:

PROOF: By the Mean Value Theorem and 1.2, since 1.1 and the hypothesis of the corollary imply that f is differentiable on $[a, b]$, so Theorem 1 implies it is continuous on $[a, b]$.

How much detail is necessary? For example, why do 1.1 and the hypothesis of the corollary, which asserts that f is differentiable on I , imply that f is differentiable on $[a, b]$? The proof is assuming the fact that a and b in the interval I implies that $[a, b]$ is a subset of I . Should this also be mentioned?

If you are writing the proof to show someone else that the theorem is correct, then the answer depends on the sophistication of the reader. A beginning student needs more help understanding a proof than does a mathematician.

If you are writing the proof for yourself to make sure that the theorem is correct, then the answer is simple: if the truth of a statement is not completely obvious, or if you suspect that there may be just the slightest possibility that it is not correct, then more detail is needed. When you write a proof, you believe the theorem to be true. The only way to avoid errors is to be ruthlessly suspicious of everything you believe. Otherwise, your natural desire to confirm what you already believe to be true will cause you to miss gaps in the proof; and every gap could hide an error that makes the entire result wrong.

Our proof of step 2 is a prose paragraph. As with any prose proof, every detail we add to it makes it harder to follow. In ordinary mathematical writing, the only solution to this problem would be to state and prove the step as a separate lemma. However, making each such subproof a lemma would submerge the interesting results in a sea of lemmas. With structured proofs there is a simple solution: replace the paragraph with a structured proof. Figure 4 shows the result of doing this for the proof of step 2 above.

2. There is some x in (a, b) with $f'(x) = \frac{f(b) - f(a)}{b - a}$.

- 2.1. f is differentiable on $[a, b]$.

PROOF: By 1.1, since f is differentiable on I by hypothesis.

- 2.2. f is continuous on $[a, b]$.

PROOF: By 2.1 and Theorem 1.

- 2.3. Q.E.D.

PROOF: By 2.1, 2.2, and the Mean Value Theorem.

Figure 4: An expanded proof of step 2.

With hypertext, there is no problem adding extra detail like this. We could add enough levels to reduce the reasoning to applications of elementary axioms. The reader can stop opening lower levels of the proof when satisfied that she understands why the statement is true. With proofs on paper, extra details kill trees. But while not as convenient as hypertext, the use of indentation makes it easy for the reader to skip over details that do not interest her.

4 A Language for Structured Proofs

TLA⁺ is a formal language designed for specifying and reasoning about algorithms and computer systems [3]. It includes a standard formalization of ordinary mathematics based on first-order logic and Zermelo-Fraenkel set theory. TLA⁺ contains constructs for writing proofs that formalize the style of structured proofs that I advocate. The TLA⁺ Toolbox is a program with a graphical interface for writing and checking TLA⁺ specifications and proofs. It provides the type of hypertext viewer of structured proofs that I expect eventually to be commonplace.

This section describes the TLA⁺ proof constructs that are relevant for ordinary mathematics. With one possible exception, these constructs can be written informally so that their meanings are obvious to a reader who has never before seen structured proofs. However, it is as silly to express logical proof structures in words as it is to express equations in words. When mathematicians leave the 17th century and begin writing structured proofs, I trust that they will adopt compact notation to replace phrases like “We now consider the case in which.”

4.1 The Proof Steps of TLA⁺

Simple Assertions

The most common type of proof step is a simple assertion. Steps 2–5 of the proof in Figure 3 are such assertions. A simple assertion is a mathematical formula. For example, the statement of the corollary can be written in TLA⁺ as:

$$(1) \quad \forall f : \forall I \in SetOfIntervals : \\ (\forall x \in I : d(f)[x] > 0) \Rightarrow IsIncreasingOn(f, I)$$

where *SetOfIntervals* is defined to be the set of all intervals of real numbers, $d(f)$ is the derivative² of f , square brackets are used for function application, \Rightarrow is logical implication, and *IsIncreasingOn* is defined by

$$\text{IsIncreasingOn}(f, I) \triangleq \forall x, y \in I : (x < y) \Rightarrow (f[x] < f[y])$$

If asked to formalize the statement of the corollary, most mathematicians would probably write something like (1), except with unimportant notational differences.

A Q.E.D. step is a simple assertion of the formula that is the proof's current goal.

ASSUME/PROVE

The statement of the corollary is actually not a simple assertion. We can replace f and I by other variables without changing the meaning of the formula (1). However, if we changed f and I just in the Corollary's statement, then Spivak's proof would make no sense because the variables f and I that appear in it would be meaningless. The statement of the corollary can be expressed in TLA⁺ as this ASSUME/PROVE statement:

```
ASSUME NEW f, NEW I ∈ SetOfIntervals, ∀x ∈ I : d(f)[x] > 0
PROVE   IsIncreasingOn(f, I)
```

This ASSUME/PROVE asserts the truth of formula (1). It also declares the goal of the corollary's proof to be the PROVE formula and allows the assumptions in the ASSUME clause to be assumed in the proof. The assumption NEW x introduces a new variable x , implicitly asserting that the conclusion is true for all values of x . The assumption NEW $x \in S$ is equivalent to the two assumptions NEW x , $x \in S$.

An ASSUME/PROVE can appear as a proof step as well as the statement of the theorem. The PROVE formula is the goal of the step's proof, and the ASSUME clause's assumptions can be used only in that proof. We can consider an ordinary assertion to be an ASSUME/PROVE with an empty ASSUME clause.

It is not obvious how to write an ASSUME/PROVE step in ordinary mathematical prose. When an assumption is introduced in a prose proof, it is assumed to hold until some unspecified later point in the proof. (One reason prose proofs are hard to understand is that it can be difficult to figure out the scope of an assumption.) As in the statement of the corollary, an

²For reasons irrelevant to ordinary mathematics, prime ('') has a special meaning in TLA⁺.

ordinary prose assertion is interpreted as an ASSUME/PROVE when it is the statement of the theorem to be proved. If it appears as a proof step, we can try indicating the ASSUME/PROVE structure by writing “If we assume ..., then we can prove ...”. That and the hierarchical structuring may be good enough to convey the intended meaning. However, it seems safer to introduce ASSUME and PROVE as keywords and explain their meaning to the reader. Fortunately, ASSUME/PROVE steps are not common in informal proofs.

SUFFICES

Step 1 in Figure 3 can be written in TLA⁺ as this SUFFICES step:

```
SUFFICES ASSUME NEW  $a \in I$ , NEW  $b \in I$ ,  $a < b$ 
      PROVE  $f[b] > f[a]$ 
```

This step asserts the truth of the formula

$$(2) \quad (\forall a, b \in I : (a < b) \Rightarrow (f[b] > f[a])) \Rightarrow \text{IsIncreasingOn}(f, I)$$

where the hypothesis of the implication (2) is the assertion of the ASSUME/PROVE, and the conclusion *IsIncreasingOn*(f , I) of (2) is the proof’s current goal. The step changes the current goal of the proof to be the PROVE formula $f[b] > f[a]$; and it allows the assumptions $a \in I$, $b \in I$, and $a < b$ of the ASSUME clause to be assumed in the rest of the proof. These assumptions do not apply to the proof of the step itself, and the NEW variables a and b are meaningless in that proof.

A proof by contradiction of a simple assertion F begins with the step

```
SUFFICES ASSUME  $\neg F$ 
      PROVE FALSE
```

A SUFFICES step can also have the form SUFFICES F for a formula F . Since a formula is an ASSUME/PROVE with no assumptions, this step asserts that F implies the proof’s current goal, and the step’s proof must prove this assertion. The step changes the current goal to F .

The SUFFICES construct is not necessary; any SUFFICES can be eliminated by restructuring the proof. For example, we could eliminate the “It suffices to” from the proof of Figure 3 by using the following top-level structure:

1. ASSUME 1. a and b are points in I .
 2. $a < b$
- ```
PROVE $f[b] > f[a]$
```

2. Q.E.D.

PROOF: By 1 and definition of an increasing function.

The proof of the original step 1 appears in the proof of the new Q.E.D step; steps 2–5 of the original proof become the proof of the new step 1.

As this example illustrates, removing a SUFFICES adds one level to the proof. Proofs are generally easiest to read if each level contains about 4 to 10 steps. The SUFFICES step eliminates the kind of two-step proof that would otherwise be needed to prove the corollary.

Step 1 of Figure 3 shows that a SUFFICES is easily expressed with informal prose, even if it is a SUFFICES ASSUME/PROVE.

#### PICK

If we were to expand the proof of step 4 of Figure 3, it would look like this.

4.1. Pick  $x$  in  $(a, b)$  with  $f'(x) = \frac{f(b) - f(a)}{b - a}$ .

PROOF: Such an  $x$  exists by step 2.

4.2. Q.E.D.

PROOF: By 4.1 and 3.

Step 4.1 is expressed formally by a step of the form

PICK  $x \in S : P(x)$

This step introduces the new variable  $x$  and asserts that  $P(x)$  is true. The step's proof must show that there exists an  $x$  satisfying  $P(x)$ .

There is no problem expressing PICK in prose.

#### CASE

If  $F$  is a formula, the step CASE  $F$  is an abbreviation for

ASSUME  $F$

PROVE Q.E.D.

where Q.E.D. stands for the formula that is the current goal. An ordinary proof by cases ends with a sequence of CASE steps followed by the Q.E.D. step. Here is a typical example:

1. PICK  $n \in S : \dots$
2. CASE  $n \geq 0$
3. CASE  $n < 0$

#### 4. Q.E.D.

PROOF: By 1, 2, and 3.

In general, the Q.E.D. step's proof cites the CASE statements and shows that the cases are exhaustive—which in this example is presumed to be trivial because  $S$  is a set of numbers.

It is easy to express a CASE statement in prose—for example:

We now consider the case in which  $n \geq 0$ .

However, why not just write CASE  $n \geq 0$ ? Readers will understand what it means.

#### Definitions

It is often convenient to give some expression a name in part of the proof. TLA<sup>+</sup> allows definitions as proof steps. A single step can contain multiple definitions. The definition is in effect for the rest of the proof's current level.

### 4.2 Hierarchical Numbering

Figure 4 introduced a naming scheme in which, for example, 3.3.4 is the 4<sup>th</sup> step in the proof of step 3.3. The most obvious problem with this scheme is that step names get longer as the proof gets deeper. For steps beyond level 4, the names are too hard to read and take up too much space.

A less obvious problem is that any step can be mentioned anywhere in the proof. One can refer to step 2.4.1 from inside the proof of step 3.3.4. Such a reference should not be allowed because it violates the hierarchical structuring of the proof. It is illegal to use step 2.4.1 in the proof of step 3.3.4 if step 2 or step 2.4 is an ASSUME/PROVE, because step 2.4.1 has then been proved under assumptions that may not hold for the proof of 3.3.4. Even if there is no ASSUME/PROVE making the reference illegal, such a reference makes the proof harder to read. The proof of step 3.3.4 should refer only to the following steps: 1, 2, 3.1, 3.2, 3.3.1, 3.3.2, and 3.3.3.

In TLA<sup>+</sup>, the 4<sup>th</sup> step of a level 3 proof is named  $\langle 3 \rangle 4$ . A proof can have many different steps named  $\langle 3 \rangle 4$ . However, at any point in the proof, only one of them can be referred to without violating the hierarchical structure. (Proving this is a nice little exercise.) Any valid reference to step  $\langle 3 \rangle 4$  refers to the most recent preceding step with that name. This numbering scheme is used in the TLA<sup>+</sup> proof of Spivak's corollary in the appendix.

PROOF:  $\Pi = \mathcal{C}(\Pi \wedge L_1)$  [Hypothesis 1]  
 $\subseteq \mathcal{C}(\Pi \wedge L_2)$  [Hypothesis 2 and monotonicity of closure]  
 $\subseteq \mathcal{C}(\Pi)$  [monotonicity of closure]  
 $= \Pi$  [Hypothesis 1, which implies  $\Pi$  closed]

This proves that  $\Pi = \mathcal{C}(\Pi \wedge L_2)$ .

Figure 5: An example of equational reasoning.

### 4.3 Equational Proofs

A different kind of structuring is provided by equational reasoning. An equational proof consists of a sequence of relations and their proofs, from which one deduces a relation by transitivity—for example, we prove

$$(3) \quad a + 1 \leq b^2 - 3 = \sqrt{c - 42} \leq d$$

and deduce  $a + 1 \leq d$ . I find such proofs to be elegant, and I use them when I can. Figure 5 shows a more sophisticated type of equational reasoning using set inclusion; it is a slightly modified version of a proof from a paper of which I was an author [1].

Equational proofs have a serious drawback when printed on paper: there seems to be no good way to display hierarchical proofs of the individual relations. On paper, equational reasoning works well only as a lowest-level proof, where the proof of each relation is very short—as in Figure 5. TLA<sup>+</sup> provides the following way of writing the sequence of relations (3) in a proof:

$$\begin{aligned} \langle 1 \rangle 1. \quad & a + 1 \leq b^2 - 3 \\ \langle 1 \rangle 2. \quad & @ = \sqrt{c - 42} \\ \langle 1 \rangle 3. \quad & @ \leq d \end{aligned}$$

where the @ symbol stands for the preceding expression. However, this lacks the visual simplicity of (3). There is no problem displaying hierarchically structured equational proofs with hypertext.

### 4.4 Comments

Like any computer-readable language, TLA<sup>+</sup> allows comments to aid human readers. With hypertext, comments can be attached to any part of a proof, to be popped up and hidden as the reader wishes. On paper, comments in arbitrary places can be distracting. However, there is one form of comment that works well on paper for both formal and informal proofs: a proof sketch that comes between a statement and its proof. The sketch can explain the

intuition behind the proof and can point out the key steps. Proof sketches can be used at any level in a hierarchical proof, including before the highest-level proof. A reader not interested in the details of that part of the proof can read just the proof sketch and skip the steps and their proofs.

## 4.5 Completely Formal Proofs

In principle, the proof of a theorem should show that the theorem can be formally deduced from axioms by the application of proof rules. In practice, we never carry a proof down to that level of detail. However, a mathematician should always be able to keep answering the question *why?* about a proof, all the way down to the level of axioms. A completely formal proof is the Platonic ideal.

Most mathematicians have no idea how easy it is to formalize mathematics. Their image of formalism is the incomprehensible sequences of symbols found in *Principia Mathematica*. The appendix contains formal TLA<sup>+</sup> definitions of intervals, limits, continuity, and the derivative, assuming only the definitions of the real numbers and of ordinary arithmetic operations. I expect most readers will be surprised to learn that this takes only 19 lines. The appendix also contains a TLA<sup>+</sup> proof of Spivak's corollary that has been checked by the TLAPS proof system [6].

Formalizing mathematics is easy, but writing formal, machine-checkable proofs is not. It will be decades before mechanical proof checkers are good enough that writing a machine-checked proof is no harder than writing a careful informal proof. Until then, there is little reason for a mathematician to write formal mathematics.

However, there is good reason for teaching how to write a formal proof as part of a standard mathematics education. Mathematicians think that the logic of the proofs they write is completely obvious, but our examination of Spivak's proof shows that they are wrong. Students are expected to learn how to write logically correct proofs from examples that, when read literally, are illogical. (Recall the first sentence of Spivak's proof.) It is little wonder that so few of them succeed. Learning to write structured formal proofs that a computer can check will teach students what a proof is. Going from these formal proofs to structured informal proofs would then be a natural step.

Is it crazy to think that students who cannot learn to write proofs in prose can learn to write them in an unfamiliar formal language and get a computer to check them? Anyone who finds it crazy should consider how many students learn to write programs in unfamiliar formal languages and

get a computer to execute them, and how few now learn to write proofs.

For reasons mentioned in the appendix, TLA<sup>+</sup> and its TLAPS prover are not ideal for teaching mathematics students. However, the structured proofs of TLA<sup>+</sup> make it the best currently available language for the task that I know of. It should be satisfactory for writing proofs in some particular domain such as elementary group theory.

## 5 Experience

I am a computer scientist who was educated as a mathematician. I discovered structured proofs through my work on concurrent (multiprocess) algorithms. These algorithms can be quite subtle and hard to get right; their correctness proofs require a degree of precision and rigor unknown to most mathematicians (and many computer scientists). A missing hypothesis, such as that a set must be nonempty, which is a trivial omission in a mathematical theorem, can mean a serious bug in an algorithm.

Proofs of algorithms are most often mathematically shallow but complicated, requiring many details to be checked. With traditional prose proofs, I found it impossible to make sure that I had not simply forgotten to check some detail. Computer science offers a standard way to handle complexity: hierarchical structure. Structured proofs were therefore an obvious solution. They worked so well for proofs of algorithms that I tried them on the more mathematical proofs that I write. I have used them for almost every proof of more than about ten lines that I have published since 1991. (The only exceptions I can find are in a paper in which the proofs served only to illustrate how certain formal proof rules are used.)

My earlier paper on structured proofs described how effective they are at catching errors. It recounted how only by writing such a proof was I able to re-discover an error in a proof of the Schroeder-Bernstein theorem in a well-known topology text [2, page 28]. I recently received email from a mathematician saying that he had tried unsuccessfully to find that error by writing a structured proof. I asked him to send me his proof, and he responded:

I tried typing up the proof that I'd hand-written, and in the process, I think I've found the fundamental error... I now really begin to understand what you mean about the power of this method, even if it did take me hours to get to this point!

It is instructive that, to find the error, he had to re-write his proof to be

read by someone else. Eliminating errors requires care. Structured proofs make it possible, not inevitable.

Over the years, I have published quite a few papers with structured proofs. For proofs with the level of detail typical of mathematics papers, I cannot remember any reader commenting on the proof style unless explicitly asked to. Structured proofs are easy enough to read that one forgets about the form and concentrates on the content. However, I have also published some papers with long, excruciatingly detailed proofs. I am reluctant to publish a paper with a short proof if I would not have been able to find the correct result without writing a longer one. Here are a referee's comments on one such proof.

The proofs... are lengthy, and are presented in a style which I find very tedious. I think the readers... are going to be more interested in understanding the techniques and how they can apply them, than they will be in reading the formal proofs. A problem with the proofs is that they do not clearly distinguish the trivial manipulations from the nontrivial or surprising parts. ... My feeling is that informal proof sketches... to explain the crucial ideas in each result would be more appropriate.

I think the referee would have found the proofs much more tedious if written in a conventional prose style, but my co-author and I could have made the proof easier to read had we used the proof sketches described in Section 4.4 above. I do agree that the proofs were too long and detailed for the journal's readers. Today, the obvious approach would be to put the long proof on the Web and publish a proof sketch with a link to the real proof. The Web was in its infancy when the paper was published, but the editor agreed to publish the proofs as a separate appendix available only on-line. None of the first reviewers had read the proofs, so the editor found another referee to do that. When asked how he or she found the proof style, the referee responded:

I have found the hierarchical structuring of proofs to be very helpful, if read top-down according to the suggestions of the authors. In fact, it might well be the only way to present long proofs... in a way that is both detailed (to ensure correctness) and readable. For long proofs, I think that describing the idea of the proof in a few words at the beginning (if appropriate) would help make them more understandable. ... But in general, I found the structured approach very effective.

## 6 Objections to Structured Proof

When lecturing about structured proofs, I have heard many objections to them. I cannot recall any objection that I found to be based on a rational argument; they have all been essentially emotional. Here are three common ones.

### **They are too complicated.**

In lectures, I usually flash on the screen one of my multi-page structured proofs. People have reacted by saying that the structuring makes the proof too complicated, as if replacing the numbering and indentation by prose would magically simplify the proof. One mathematician described how she had explained a beautiful little proof by Hardy to an audience of non-mathematicians, and that the audience could not possibly understand my proofs. She apparently believed that structuring Hardy's tiny proof would turn it into multiple pages full of obscure symbols.

It is an unfortunate fact that being rigorous requires filling in missing details, which makes a proof longer. As we saw with Spivak's corollary, a structured proof makes it easier to see what is missing; this would lead mathematicians to correct the omissions, resulting in longer proofs. Fortunately, structuring allows us to add those details without making the proof any harder to read as hypertext, and only a little harder to read on paper.

### **They don't explain why the proof works.**

Mathematicians seem to think that their proofs explain themselves. I cannot remember reading a mathematician's proof that was both a proof and an explanation of why the proof works. It's hard enough to make the structure of a prose proof clear; doing it while also providing an intuitive explanation is a formidable task.

I suspect that this objection is based on confusing a proof sketch with a proof. Proof sketches are fine, but they are not proofs. Mathematicians sometimes precede a proof with a proof sketch, but there is no easy way to relate the steps of the proof with the proof sketch. The ability to add proof sketches at any level of a structured proof makes it possible to provide a much clearer explanation of why a proof works.

## A proof should be great literature.

This is nonsense. A proof should not be great literature; it should be beautiful mathematics. Its beauty lies in its logical structure, not in its prose. Proofs are more like architecture than like literature, and architects do not use prose to design buildings. Prose cannot add to the beauty of  $e^{i\pi} + 1 = 0$ , and it is a poor medium for expressing the beauty of a proof.

## 7 Beginning

When I started writing structured proofs, I quickly found them to be completely natural. Writing non-trivial prose proofs now seems as archaic to me as writing

The number  $e$  raised to the power of  $i$  times  $\pi$ , when added to 1, equals 0.

Imagine how many errors we would make performing algebraic calculations with equations written in prose. Writing proofs in prose is equally error prone. As I reported in my earlier paper, anecdotal evidence indicates that a significant fraction of published mathematics contains serious errors. This will not change until mathematicians understand that precision and rigor, not prose, are the natural medium of mathematics, and they stop writing 17<sup>th</sup> century proofs.

Fortunately, it's not hard to write 21<sup>st</sup> century proofs. There is no need to wait until other mathematicians are doing it. You can begin by just adding structure to an existing proof, as we did with Spivak's proof. Start by rewriting a simple proof and then try longer ones. You should soon find this a much more logical way to write your proofs, and readers will have no trouble understanding the proof style.

Writing structured proofs is liberating. It allows you to concentrate on logical structure instead of sentence structure. You will no longer waste your time searching for different ways to say *therefore*. To help you typeset your structured proofs, a L<sup>A</sup>T<sub>E</sub>X package and an associated computer program are available on the Web [4].

## Acknowledgement

In his courses on analysis and algebraic topology, Richard Palais taught me how mathematics could be made precise and rigorous, and thereby more beautiful.

## References

- [1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
- [2] John L. Kelley. *General Topology*. The University Series in Higher Mathematics. D. Van Nostrand Company, Princeton, New Jersey, 1955.
- [3] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL <http://lamport.org>. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.
- [4] Leslie Lamport. Useful LaTeX packages. <http://research.microsoft.com/en-us/um/people/lamport/latex/latex.html>. The page can also be found by searching the Web for the 23-letter string formed by concatenating `uid` and `lamportlatexpackages`.
- [5] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., 1993. A symposium in honor of Richard Palais’ sixtieth birthday. Also published in *American Mathematical Monthly*, 102(7):600–608, August–September 1995.
- [6] Microsoft Research-INRIA Joint Centre. Tools and methodologies for formal specifications and for proofs. <http://www.msr-inria.inria.fr/Projects/tools-for-formal-specs>.
- [7] Michael Spivak. *Calculus*. W. A. Benjamin, Inc., New York, 1967.

## Appendix: A Formal Proof

This appendix contains a TLA<sup>+</sup> formalization of Spivak's proof, preceded by formal definitions of the necessary concepts of differential calculus and statements of the two theorems used in that proof. This is all done in a TLA<sup>+</sup> module named *Calculus*.

Mathematicians will be struck by the absence of some common mathematical notation. For example, the open interval  $(a, b)$  is written *OpenInterval*( $a, b$ ). No single syntax can capture the wide variety of notation used in mathematics, where  $(a, b)$  may be an interval or an ordered pair, depending on the context. A good formal language for math should permit such context-dependent notation. TLA<sup>+</sup> was not designed for use by mathematicians, so it provides conventional notation mainly for basic operators of logic and set theory.

TLAPS uses sophisticated algorithms for performing simple reasoning about integers. However, because it is still under development and real numbers are seldom used in algorithms, TLAPS does not yet provide any special support for reasoning about reals. The proof of Spivak's corollary therefore assumes without proof five very simple facts about real numbers.

The appendix shows the typeset version of the *Calculus* module, not its actual ASCII text. For example, the definition of *OpenInterval* that is typeset as

$$\text{OpenInterval}(a, b) \triangleq \{r \in \text{Real} : (a < r) \wedge (r < b)\}$$

appears in the module as

```
OpenInterval(a, b) == {r \in Real : (a < r) /\ (r < b)}
```

The L<sup>A</sup>T<sub>E</sub>X source for the typeset version was generated from the ASCII by a program, but it is possible that editing of the document introduced errors.

The rest of the appendix consists of the *Calculus* module together with interspersed comments explaining some of the TLA<sup>+</sup> notation. These explanations and a thorough knowledge of calculus should allow you to figure out what's going on. However, you probably won't find it easy reading. This is hardly surprising, since the definition of the derivative, which is line 20 of the module, appears on page 127 of Spivak's book. With a few pages of explanation, I expect a mathematician would find the TLA<sup>+</sup> formulas as easy to read as Spivak's text.

The following EXTENDS statement imports the standard *Reals* module that defines the set *Real* of real numbers and the usual operators on real numbers such as / (division) and  $\leq$ .

EXTENDS *Reals*

$$\text{OpenInterval}(a, b) \triangleq \{r \in \text{Real} : (a < r) \wedge (r < b)\}$$

$$\text{ClosedInterval}(a, b) \triangleq \{r \in \text{Real} : (a \leq r) \wedge (r \leq b)\}$$

In TLA<sup>+</sup>, SUBSET *S* is the power set (the set of all subsets) of *S*.

$$\text{SetOfIntervals} \triangleq \{S \in \text{SUBSET Real} : \forall x, y \in S : \text{OpenInterval}(x, y) \subseteq S\}$$

UNION *S* is the union of all sets that are elements of the set *S*, and  $[S \rightarrow T]$  is the set of functions with domain *S* and range a subset of *T*. Hence, the following defines *RealFunction* to be the set of all real-valued functions whose domain is a set of real numbers.

$$\text{RealFunction} \triangleq \text{UNION } \{[S \rightarrow \text{Real}] : S \in \text{SUBSET Real}\}$$

$$\text{AbsoluteValue}(a) \triangleq \text{IF } a > 0 \text{ THEN } a \text{ ELSE } -a$$

$$\text{OpenBall}(a, e) \triangleq \{x \in \text{Real} : e > \text{AbsoluteValue}(x - a)\}$$

$$\text{PositiveReal} \triangleq \{r \in \text{Real} : r > 0\}$$

TLA<sup>+</sup> uses square brackets for function application, as in  $f[x]$ . Parentheses are reserved for arguments of defined operators like *OpenInterval*.

$$\begin{aligned} \text{IsLimitAt}(f, a, b) &\triangleq (b \in \text{Real}) \wedge \\ &\quad \forall e \in \text{PositiveReal} : \exists d \in \text{PositiveReal} : \\ &\quad \quad \forall x \in \text{OpenBall}(a, d) \setminus \{a\} : f[x] \in \text{OpenBall}(b, e) \end{aligned}$$

$$\text{IsContinuousAt}(f, a) \triangleq \text{IsLimitAt}(f, a, f[a])$$

$$\text{IsContinuousOn}(f, S) \triangleq \forall x \in S : \text{IsContinuousAt}(f, x)$$

Mathematics provides no formal notation for writing a function. In TLA<sup>+</sup>,  $[x \in S \mapsto e(x)]$  is the function with domain *S* that maps *x* to *e(x)* for every *x* in *S*. If *f* is a function, then DOMAIN *f* is its domain.

$$\begin{aligned} \text{IsDerivativeAt}(f, a, b) &\triangleq \\ &\exists e \in \text{PositiveReal} : \\ &\quad (\text{OpenBall}(a, e) \subseteq \text{DOMAIN } f) \wedge \\ &\quad \text{IsLimitAt}([x \in \text{OpenBall}(a, e) \setminus \{a\} \mapsto (f[x] - f[a])/(x - a)], a, b) \end{aligned}$$

$$\text{IsDifferentiableAt}(f, a) \triangleq \exists b \in \text{Real} : \text{IsDerivativeAt}(f, a, b)$$

$$\text{IsDifferentiableOn}(f, S) \triangleq \forall x \in S : \text{IsDifferentiableAt}(f, x)$$

CHOOSE  $x : P(x)$  is an arbitrary value  $x$  satisfying  $P(x)$ , if such a value exists; otherwise its value is unspecified. The CHOOSE operator is known to logicians as Hilbert's  $\varepsilon$ .

$$d(f) \triangleq [x \in \text{DOMAIN } f \mapsto \text{CHOOSE } y : \text{IsDerivativeAt}(f, x, y)]$$

The following THEOREM asserts the truth of the formula  $\forall f : \dots$  and names that formula *Theorem1*.

$$\begin{aligned} \text{THEOREM } \text{Theorem1} &\triangleq \forall f \in \text{RealFunction} : \forall a \in \text{Real} : \\ &\quad \text{IsDifferentiableAt}(f, a) \Rightarrow \text{IsContinuousAt}(f, a) \end{aligned}$$

$$\text{IsIncreasingOn}(f, I) \triangleq \forall x, y \in I : (x < y) \Rightarrow (f[x] < f[y])$$

$$\begin{aligned} \text{THEOREM } \text{MeanValueTheorem} &\triangleq \\ \forall f \in \text{RealFunction} : \forall a, b \in \text{Real} : & \\ ( & (a < b) \\ & \wedge \text{IsContinuousOn}(f, \text{ClosedInterval}(a, b)) \\ & \wedge \text{IsDifferentiableOn}(f, \text{OpenInterval}(a, b))) \\ \Rightarrow & (\exists x \in \text{OpenInterval}(a, b) : \\ & d(f)[x] = (f[b] - f[a])/(b - a)) \end{aligned}$$

We assume without proof the following five trivial facts about real numbers. TLAPS easily proves the first four for integers.

$$\text{PROPOSITION } \text{Fact1} \triangleq \forall x \in \text{Real} : x - x = 0$$

$$\text{PROPOSITION } \text{Fact2} \triangleq \forall x, y \in \text{Real} : (x \leq y) \equiv (x < y) \vee (x = y)$$

$$\text{PROPOSITION } \text{Fact3} \triangleq \forall x, y \in \text{Real} : x - y \in \text{Real}$$

$$\text{PROPOSITION } \text{Fact4} \triangleq \forall x, y \in \text{Real} : (x < y) \equiv (y - x > 0)$$

$$\text{PROPOSITION } \text{Fact5} \triangleq \forall x, y \in \text{Real} : (y > 0) \wedge (x/y > 0) \Rightarrow (x > 0)$$

Below is the corollary and its proof. Each step of the high-level proof formalizes the correspondingly-numbered step of the proof in Figure 3, where step  $\langle 1 \rangle 1a$  has been added in the formal proof. The proof of step  $\langle 1 \rangle 2$  formalizes the proof of Figure 4.

The lowest-level paragraphs of an informal proof are replaced with BY statements that say what facts and definitions (DEF) are used. (Assumptions in NEW declarations and in the statement of the corollary are automatically used by TLAPS.) The proof has been decomposed for the benefit of TLAPS, not for a human reader.

$$\begin{aligned} \text{COROLLARY } \text{Spivak} &\triangleq \text{ASSUME NEW } f \in \text{RealFunction}, \\ &\quad \text{NEW } I \in \text{SetOfIntervals}, \\ &\quad \text{IsDifferentiableOn}(f, I), \\ &\quad \forall x \in I : d(f)[x] > 0 \\ \text{PROVE } &\text{IsIncreasingOn}(f, I) \end{aligned}$$

$\langle 1 \rangle 1.$  SUFFICES ASSUME NEW  $a \in I$ , NEW  $b \in I$ ,  $a < b$   
 PROVE  $f[a] < f[b]$   
 BY DEF *IsIncreasingOn*

$\langle 1 \rangle 1a.$   $(OpenInterval(a, b) \subseteq I)$   
 $\wedge (ClosedInterval(a, b) \subseteq I)$   
 $\wedge (OpenInterval(a, b) \subseteq ClosedInterval(a, b))$   
 $\langle 2 \rangle 1.$   $OpenInterval(a, b) \subseteq I$   
 BY DEF *SetOfIntervals*  
 $\langle 2 \rangle 2.$   $ClosedInterval(a, b) = OpenInterval(a, b) \cup \{a, b\}$   
 BY  $\langle 1 \rangle 1$ , *Fact2* DEF *ClosedInterval*, *OpenInterval*, *SetOfIntervals*  
 $\langle 2 \rangle 3.$  QED  
 BY  $\langle 2 \rangle 1$ ,  $\langle 2 \rangle 2$

$\langle 1 \rangle 2.$   $\exists x \in OpenInterval(a, b) : d(f)[x] = (f[b] - f[a])/(b - a)$   
 $\langle 2 \rangle 1.$  *IsDifferentiableOn*( $f$ ,  $ClosedInterval(a, b)$ )  
 BY  $\langle 1 \rangle 1a$ ,  $\langle 1 \rangle 1$  DEF *IsDifferentiableOn*  
 $\langle 2 \rangle 2.$  *IsContinuousOn*( $f$ ,  $ClosedInterval(a, b)$ )  
 $\langle 3 \rangle 1.$  SUFFICES ASSUME NEW  $x \in ClosedInterval(a, b)$   
 PROVE *IsContinuousAt*( $f$ ,  $x$ )  
 BY DEF *IsContinuousOn*  
 $\langle 3 \rangle 2.$  *IsDifferentiableAt*( $f$ ,  $x$ )  
 BY  $\langle 2 \rangle 1$  DEF *IsDifferentiableOn*  
 $\langle 3 \rangle 3.$  QED  
 BY  $\langle 3 \rangle 2$ , *Theorem1* DEF *ClosedInterval*  
 $\langle 2 \rangle 3.$  QED  
 BY  $\langle 1 \rangle 1$ ,  $\langle 1 \rangle 1a$ ,  $\langle 2 \rangle 1$ ,  $\langle 2 \rangle 2$ , *MeanValueTheorem*  
 DEF *SetOfIntervals*, *IsDifferentiableOn*

$\langle 1 \rangle 3.$   $\forall x \in OpenInterval(a, b) : d(f)[x] > 0$   
 BY  $\langle 1 \rangle 1a$

$\langle 1 \rangle 4.$   $(f[b] - f[a])/(b - a) > 0$   
 $\langle 2 \rangle 1.$  PICK  $x \in OpenInterval(a, b) :$   
 $d(f)[x] = (f[b] - f[a])/(b - a)$   
 BY  $\langle 1 \rangle 2$   
 $\langle 2 \rangle 2.$   $d(f)[x] > 0$   
 BY  $\langle 2 \rangle 1$  DEF *SetOfIntervals*  
 $\langle 2 \rangle 3.$  QED  
 BY  $\langle 2 \rangle 1$ ,  $\langle 2 \rangle 2$

⟨1⟩5. QED  
 ⟨2⟩1.  $(a \in \text{Real}) \wedge (b \in \text{Real})$   
     BY DEF *SetOfIntervals*  
 ⟨2⟩2.  $(f[a] \in \text{Real}) \wedge (f[b] \in \text{Real})$   
 ⟨3⟩1. SUFFICES ASSUME NEW  $x \in \text{Real}$ ,  
         *IsDifferentiableAt*( $f$ ,  $x$ )  
         PROVE  $f[x] \in \text{Real}$   
     BY ⟨2⟩1 DEF *IsDifferentiableOn*  
 ⟨3⟩2.  $\forall e \in \text{PositiveReal} : x \in \text{OpenBall}(x, e)$   
     BY *Fact1* DEF *OpenBall*, *PositiveReal*, *AbsoluteValue*  
 ⟨3⟩3.  $x \in \text{DOMAIN } f$   
     BY ⟨3⟩1, ⟨3⟩2 DEF *IsDifferentiableAt*, *IsDerivativeAt*  
 ⟨3⟩4. QED  
     BY ⟨3⟩3 DEF *IsDifferentiableAt*, *IsDerivativeAt*, *RealFunction*  
 ⟨2⟩3.  $(f[b] - f[a] \in \text{Real}) \wedge (b - a \in \text{Real})$   
     BY ⟨2⟩1, ⟨2⟩2, *Fact3*  
 ⟨2⟩4.  $f[b] - f[a] > 0$   
     BY ⟨1⟩1, ⟨1⟩4, ⟨2⟩1, ⟨2⟩3, *Fact4*, *Fact5*  
 ⟨2⟩5. QED  
     BY ⟨2⟩2, ⟨2⟩4, *Fact4*

# LATEX

*A Document Preparation System*

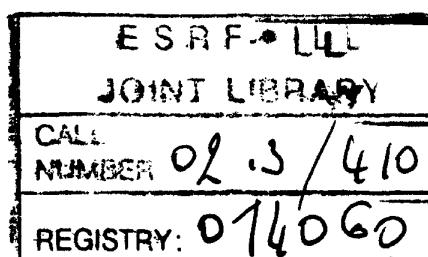
*User's Guide and Reference Manual*

*Leslie Lamport*

Digital Equipment Corporation



Illustrations by Duane Bibby



**Addison-Wesley Publishing Company**



Reading, Massachusetts • Menlo Park, California • New York  
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn • Sydney  
Singapore • Tokyo • Madrid • San Juan • Milan • Paris

Many of the designations used by the manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The procedures and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

**Library of Congress Cataloging-in-Publication Data**

Lamport, Leslie.

**LATEX** : a document preparation system / Leslie Lamport. -- 2nd ed.  
p. cm.

Includes bibliographical references and index.

ISBN 0-201-52983-1

1. **LATEX** (Computer file) 2. Computerized typesetting. I. Title.

Z253.4.L38L35 1994

682.2 ' 2544536-dc20

93-39691

CIP

This documentation was prepared with **LATEX** and reproduced by Addison-Wesley from camera-ready copy supplied by the author.

This book describes **LATEX** 2<sub>ε</sub>, the second widely released version of **LATEX**.

Reprinted with corrections November, 1994

Copyright © 1994, 1985 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher. Printed in the United States of America.

2 3 4 5 6 7 8 9 10-CRW-97969594

*To Ellen*



# Contents

|                                              |           |
|----------------------------------------------|-----------|
| <b>Preface</b>                               | <b>xv</b> |
| <b>1 Getting Acquainted</b>                  | <b>1</b>  |
| 1.1 How to Avoid Reading This Book . . . . . | 2         |
| 1.2 How to Read This Book . . . . .          | 3         |
| 1.3 The Game of the Name . . . . .           | 5         |
| 1.4 Turning Typing into Typography . . . . . | 5         |
| 1.5 Why $\text{\LaTeX}$ ? . . . . .          | 7         |
| 1.6 Turning Ideas into Input . . . . .       | 8         |
| 1.7 Trying It Out . . . . .                  | 8         |
| <b>2 Getting Started</b>                     | <b>11</b> |
| 2.1 Preparing an Input File . . . . .        | 12        |
| 2.2 The Input . . . . .                      | 13        |
| 2.2.1 Sentences and Paragraphs . . . . .     | 13        |
| Quotation Marks . . . . .                    | 13        |
| Dashes . . . . .                             | 14        |
| Space After a Period . . . . .               | 14        |
| Special Symbols . . . . .                    | 15        |
| Simple Text-Generating Commands . . . . .    | 15        |
| Emphasizing Text . . . . .                   | 16        |
| Preventing Line Breaks . . . . .             | 17        |
| Footnotes . . . . .                          | 17        |
| Formulas . . . . .                           | 18        |
| Ignorable Input . . . . .                    | 19        |
| 2.2.2 The Document . . . . .                 | 19        |
| The Document Class . . . . .                 | 19        |
| The Title “Page” . . . . .                   | 20        |
| 2.2.3 Sectioning . . . . .                   | 21        |
| 2.2.4 Displayed Material . . . . .           | 23        |
| Quotations . . . . .                         | 24        |
| Lists . . . . .                              | 24        |

|                                                       |           |
|-------------------------------------------------------|-----------|
| Poetry . . . . .                                      | 25        |
| Displayed Formulas . . . . .                          | 26        |
| 2.2.5 Declarations . . . . .                          | 27        |
| 2.3 Running L <sup>A</sup> T <sub>E</sub> X . . . . . | 28        |
| 2.4 Helpful Hints . . . . .                           | 31        |
| 2.5 Summary . . . . .                                 | 32        |
| <b>3 Carrying On</b> . . . . .                        | <b>35</b> |
| 3.1 Changing the Type Style . . . . .                 | 36        |
| 3.2 Symbols from Other Languages . . . . .            | 38        |
| 3.2.1 Accents . . . . .                               | 38        |
| 3.2.2 Symbols . . . . .                               | 38        |
| 3.3 Mathematical Formulas . . . . .                   | 39        |
| 3.3.1 Some Common Structures . . . . .                | 40        |
| Subscripts and Superscripts . . . . .                 | 40        |
| Fractions . . . . .                                   | 40        |
| Roots . . . . .                                       | 40        |
| Ellipsis . . . . .                                    | 40        |
| 3.3.2 Mathematical Symbols . . . . .                  | 41        |
| Greek Letters . . . . .                               | 41        |
| Calligraphic Letters . . . . .                        | 42        |
| A Menagerie of Mathematical Symbols . . . . .         | 42        |
| Log-like Functions . . . . .                          | 44        |
| 3.3.3 Arrays . . . . .                                | 45        |
| The <i>array</i> Environment . . . . .                | 45        |
| Vertical Alignment . . . . .                          | 46        |
| More Complex Arrays . . . . .                         | 46        |
| 3.3.4 Delimiters . . . . .                            | 46        |
| 3.3.5 Multiline Formulas . . . . .                    | 47        |
| 3.3.6 Putting One Thing Above Another . . . . .       | 49        |
| Over- and Underlining . . . . .                       | 49        |
| Accents . . . . .                                     | 49        |
| Stacking Symbols . . . . .                            | 50        |
| 3.3.7 Spacing in Math Mode . . . . .                  | 50        |
| 3.3.8 Changing Style in Math Mode . . . . .           | 51        |
| Type Style . . . . .                                  | 51        |
| Math Style . . . . .                                  | 52        |
| 3.3.9 When All Else Fails . . . . .                   | 52        |
| 3.4 Defining Commands and Environments . . . . .      | 53        |
| 3.4.1 Defining Commands . . . . .                     | 53        |
| 3.4.2 Defining Environments . . . . .                 | 55        |
| 3.4.3 Theorems and Such . . . . .                     | 56        |
| 3.5 Figures and Other Floating Bodies . . . . .       | 58        |

|          |                                                      |           |
|----------|------------------------------------------------------|-----------|
| 3.5.1    | Figures and Tables . . . . .                         | 58        |
| 3.5.2    | Marginal Notes . . . . .                             | 59        |
| 3.6      | Lining It Up in Columns . . . . .                    | 60        |
| 3.6.1    | The <code>tabbing</code> Environment . . . . .       | 60        |
| 3.6.2    | The <code>tabular</code> Environment . . . . .       | 62        |
| 3.7      | Simulating Typed Text . . . . .                      | 63        |
| <b>4</b> | <b>Moving Information Around</b>                     | <b>65</b> |
| 4.1      | The Table of Contents . . . . .                      | 66        |
| 4.2      | Cross-References . . . . .                           | 67        |
| 4.3      | Bibliography and Citation . . . . .                  | 69        |
| 4.3.1    | Using <code>BIBTeX</code> . . . . .                  | 70        |
| 4.3.2    | Doing It Yourself . . . . .                          | 71        |
| 4.4      | Splitting Your Input . . . . .                       | 72        |
| 4.5      | Making an Index or Glossary . . . . .                | 74        |
| 4.5.1    | Compiling the Entries . . . . .                      | 74        |
| 4.5.2    | Producing an Index or Glossary by Yourself . . . . . | 75        |
| 4.6      | Keyboard Input and Screen Output . . . . .           | 76        |
| 4.7      | Sending Your Document . . . . .                      | 77        |
| <b>5</b> | <b>Other Document Classes</b>                        | <b>79</b> |
| 5.1      | Books . . . . .                                      | 80        |
| 5.2      | Slides . . . . .                                     | 80        |
| 5.2.1    | Slides and Overlays . . . . .                        | 81        |
| 5.2.2    | Notes . . . . .                                      | 83        |
| 5.2.3    | Printing Only Some Slides and Notes . . . . .        | 83        |
| 5.2.4    | Other Text . . . . .                                 | 84        |
| 5.3      | Letters . . . . .                                    | 84        |
| <b>6</b> | <b>Designing It Yourself</b>                         | <b>87</b> |
| 6.1      | Document and Page Styles . . . . .                   | 88        |
| 6.1.1    | Document-Class Options . . . . .                     | 88        |
| 6.1.2    | Page Styles . . . . .                                | 89        |
| 6.1.3    | The Title Page and Abstract . . . . .                | 90        |
| 6.1.4    | Customizing the Style . . . . .                      | 91        |
| 6.2      | Line and Page Breaking . . . . .                     | 93        |
| 6.2.1    | Line Breaking . . . . .                              | 93        |
| 6.2.2    | Page Breaking . . . . .                              | 96        |
| 6.3      | Numbering . . . . .                                  | 97        |
| 6.4      | Length, Spaces, and Boxes . . . . .                  | 99        |
| 6.4.1    | Length . . . . .                                     | 99        |
| 6.4.2    | Spaces . . . . .                                     | 101       |
| 6.4.3    | Boxes . . . . .                                      | 103       |
|          | LR Boxes . . . . .                                   | 104       |

|                                                       |            |
|-------------------------------------------------------|------------|
| Parboxes . . . . .                                    | 104        |
| Rule Boxes . . . . .                                  | 106        |
| Raising and Lowering Boxes . . . . .                  | 107        |
| Saving Boxes . . . . .                                | 107        |
| 6.4.4 Formatting with Boxes . . . . .                 | 108        |
| 6.5 Centering and “Flushing” . . . . .                | 111        |
| 6.6 List-Making Environments . . . . .                | 112        |
| 6.6.1 The <code>list</code> Environment . . . . .     | 112        |
| 6.6.2 The <code>trivlist</code> Environment . . . . . | 115        |
| 6.7 Fonts . . . . .                                   | 115        |
| 6.7.1 Changing Type Size . . . . .                    | 115        |
| 6.7.2 Special Symbols . . . . .                       | 116        |
| <b>7 Pictures and Colors</b>                          | <b>117</b> |
| 7.1 Pictures . . . . .                                | 118        |
| 7.1.1 The <code>picture</code> Environment . . . . .  | 119        |
| 7.1.2 Picture Objects . . . . .                       | 120        |
| Text . . . . .                                        | 120        |
| Boxes . . . . .                                       | 120        |
| Straight Lines . . . . .                              | 122        |
| Arrows . . . . .                                      | 123        |
| Stacks . . . . .                                      | 123        |
| Circles . . . . .                                     | 124        |
| Ovals and Rounded Corners . . . . .                   | 124        |
| Framing . . . . .                                     | 125        |
| 7.1.3 Curves . . . . .                                | 125        |
| 7.1.4 Grids . . . . .                                 | 126        |
| 7.1.5 Reusing Objects . . . . .                       | 127        |
| 7.1.6 Repeated Patterns . . . . .                     | 127        |
| 7.1.7 Some Hints on Drawing Pictures . . . . .        | 128        |
| 7.2 The <code>graphics</code> Package . . . . .       | 129        |
| 7.3 Color . . . . .                                   | 131        |
| <b>8 Errors</b>                                       | <b>133</b> |
| 8.1 Finding the Error . . . . .                       | 134        |
| 8.2 <code>LATEX</code> ’s Error Messages . . . . .    | 136        |
| 8.3 <code>TEX</code> ’s Error Messages . . . . .      | 140        |
| 8.4 <code>LATEX</code> Warnings . . . . .             | 145        |
| 8.5 <code>TEX</code> Warnings . . . . .               | 147        |
| <b>A Using <code>MakeIndex</code></b>                 | <b>149</b> |
| A.1 How to Use <code>MakeIndex</code> . . . . .       | 150        |
| A.2 How to Generate Index Entries . . . . .           | 150        |
| A.2.1 When, Why, What, and How to Index . . . . .     | 150        |

---

|          |                                                   |            |
|----------|---------------------------------------------------|------------|
| A.2.2    | The Basics . . . . .                              | 151        |
| A.2.3    | The Fine Print . . . . .                          | 153        |
| A.3      | Error Messages . . . . .                          | 154        |
| <b>B</b> | <b>The Bibliography Database</b>                  | <b>155</b> |
| B.1      | The Format of the <code>bib</code> File . . . . . | 156        |
| B.1.1    | The Entry Format . . . . .                        | 156        |
| B.1.2    | The Text of a Field . . . . .                     | 157        |
|          | Names . . . . .                                   | 157        |
|          | Titles . . . . .                                  | 158        |
| B.1.3    | Abbreviations . . . . .                           | 158        |
| B.1.4    | Cross-References . . . . .                        | 159        |
| B.2      | The Entries . . . . .                             | 160        |
| B.2.1    | Entry Types . . . . .                             | 160        |
| B.2.2    | Fields . . . . .                                  | 162        |
| <b>C</b> | <b>Reference Manual</b>                           | <b>165</b> |
| C.1      | Commands and Environments . . . . .               | 166        |
| C.1.1    | Command Names and Arguments . . . . .             | 166        |
| C.1.2    | Environments . . . . .                            | 167        |
| C.1.3    | Fragile Commands . . . . .                        | 167        |
| C.1.4    | Declarations . . . . .                            | 168        |
| C.1.5    | Invisible Commands and Environments . . . . .     | 169        |
| C.1.6    | The <code>\`</code> Command . . . . .             | 169        |
| C.2      | The Structure of the Document . . . . .           | 170        |
| C.3      | Sentences and Paragraphs . . . . .                | 170        |
| C.3.1    | Making Sentences . . . . .                        | 170        |
| C.3.2    | Making Paragraphs . . . . .                       | 171        |
| C.3.3    | Footnotes . . . . .                               | 172        |
| C.3.4    | Accents and Special Symbols . . . . .             | 173        |
| C.4      | Sectioning and Table of Contents . . . . .        | 174        |
| C.4.1    | Sectioning Commands . . . . .                     | 174        |
| C.4.2    | The Appendix . . . . .                            | 175        |
| C.4.3    | Table of Contents . . . . .                       | 175        |
| C.4.4    | Style Parameters . . . . .                        | 176        |
| C.5      | Classes, Packages, and Page Styles . . . . .      | 176        |
| C.5.1    | Document Class . . . . .                          | 176        |
| C.5.2    | Packages . . . . .                                | 178        |
| C.5.3    | Page Styles . . . . .                             | 179        |
| C.5.4    | The Title Page and Abstract . . . . .             | 181        |
| C.6      | Displayed Paragraphs . . . . .                    | 183        |
| C.6.1    | Quotations and Verse . . . . .                    | 184        |
| C.6.2    | List-Making Environments . . . . .                | 184        |

|        |                                                                        |     |
|--------|------------------------------------------------------------------------|-----|
| C.6.3  | The list and <code>trivlist</code> Environments . . . . .              | 185 |
| C.6.4  | <code>Verbatim</code> . . . . .                                        | 186 |
| C.7    | Mathematical Formulas . . . . .                                        | 187 |
| C.7.1  | Math Mode Environments . . . . .                                       | 187 |
| C.7.2  | Common Structures . . . . .                                            | 189 |
| C.7.3  | Mathematical Symbols . . . . .                                         | 189 |
| C.7.4  | Arrays . . . . .                                                       | 190 |
| C.7.5  | Delimiters . . . . .                                                   | 190 |
| C.7.6  | Putting One Thing Above Another . . . . .                              | 190 |
| C.7.7  | Spacing . . . . .                                                      | 191 |
| C.7.8  | Changing Style . . . . .                                               | 191 |
| C.8    | Definitions, Numbering, and Programming . . . . .                      | 192 |
| C.8.1  | Defining Commands . . . . .                                            | 192 |
| C.8.2  | Defining Environments . . . . .                                        | 192 |
| C.8.3  | Theorem-like Environments . . . . .                                    | 193 |
| C.8.4  | Numbering . . . . .                                                    | 194 |
| C.8.5  | The <code>ifthen</code> Package . . . . .                              | 195 |
| C.9    | Figures and Other Floating Bodies . . . . .                            | 197 |
| C.9.1  | Figures and Tables . . . . .                                           | 197 |
| C.9.2  | Marginal Notes . . . . .                                               | 200 |
| C.10   | Lining It Up in Columns . . . . .                                      | 201 |
| C.10.1 | The <code>tabbing</code> Environment . . . . .                         | 201 |
| C.10.2 | The <code>array</code> and <code>tabular</code> Environments . . . . . | 204 |
| C.11   | Moving Information Around . . . . .                                    | 207 |
| C.11.1 | Files . . . . .                                                        | 207 |
| C.11.2 | Cross-References . . . . .                                             | 209 |
| C.11.3 | Bibliography and Citation . . . . .                                    | 209 |
| C.11.4 | Splitting the Input . . . . .                                          | 210 |
| C.11.5 | Index and Glossary . . . . .                                           | 211 |
|        | Producing an Index . . . . .                                           | 211 |
|        | Compiling the Entries . . . . .                                        | 212 |
| C.11.6 | Terminal Input and Output . . . . .                                    | 212 |
| C.12   | Line and Page Breaking . . . . .                                       | 213 |
| C.12.1 | Line Breaking . . . . .                                                | 213 |
| C.12.2 | Page Breaking . . . . .                                                | 214 |
| C.13   | Lengths, Spaces, and Boxes . . . . .                                   | 215 |
| C.13.1 | Length . . . . .                                                       | 215 |
| C.13.2 | Space . . . . .                                                        | 216 |
| C.13.3 | Boxes . . . . .                                                        | 217 |
| C.14   | Pictures and Color . . . . .                                           | 219 |
| C.14.1 | The <code>picture</code> Environment . . . . .                         | 219 |
|        | Picture-Mode Commands . . . . .                                        | 220 |
|        | Picture Objects . . . . .                                              | 221 |

|                                              |            |
|----------------------------------------------|------------|
| Picture Declarations . . . . .               | 223        |
| C.14.2 The <b>graphics</b> Package . . . . . | 223        |
| C.14.3 The <b>color</b> Package . . . . .    | 224        |
| C.15 Font Selection . . . . .                | 225        |
| C.15.1 Changing the Type Style . . . . .     | 225        |
| C.15.2 Changing the Type Size . . . . .      | 226        |
| C.15.3 Special Symbols . . . . .             | 226        |
| <b>D What's New</b>                          | <b>227</b> |
| <b>E Using Plain <b>TeX</b> Commands</b>     | <b>231</b> |
| <b>Bibliography</b>                          | <b>235</b> |
| <b>Index</b>                                 | <b>237</b> |



---

# List of Tables

|      |                           |    |
|------|---------------------------|----|
| 3.1  | Accents.                  | 38 |
| 3.2  | Non-English Symbols.      | 39 |
| 3.3  | Greek Letters.            | 41 |
| 3.4  | Binary Operation Symbols. | 42 |
| 3.5  | Relation Symbols.         | 43 |
| 3.6  | Arrow Symbols             | 43 |
| 3.7  | Miscellaneous Symbols.    | 43 |
| 3.8  | Variable-sized Symbols.   | 44 |
| 3.9  | Log-like Functions.       | 44 |
| 3.10 | Delimiters.               | 47 |
| 3.11 | Math Mode Accents.        | 50 |

# List of Figures

|     |                                                                  |     |
|-----|------------------------------------------------------------------|-----|
| 6.1 | Boxes and how <b>T<sub>E</sub>X</b> puts them together.          | 103 |
| 6.2 | The complete definition of the <b>\face</b> command.             | 110 |
| 6.3 | The format of a list.                                            | 113 |
| 7.1 | Points and their coordinates.                                    | 119 |
| 7.2 | <b>\put</b> (1.4,2.6){\line(3,-1){4.8}}                          | 122 |
| C.1 | Making footnotes without the <b>\footnote</b> command.           | 173 |
| C.2 | Sectioning and table of contents commands.                       | 174 |
| C.3 | Page style parameters.                                           | 182 |
| C.4 | An example title.                                                | 183 |
| C.5 | Writing programs with the <b>ifthen</b> package's commands.      | 196 |
| C.6 | A <b>tabbing</b> environment example.                            | 202 |
| C.7 | Examples of the <b>tabular</b> and <b>tabular*</b> environments. | 204 |
| C.8 | A sample <b>picture</b> environment.                             | 220 |



# Preface

The first edition of this book appeared in 1985. It described  $\text{\LaTeX} 2.09$ , the first widely used version of  $\text{\LaTeX}$ . Since then,  $\text{\LaTeX}$  has become extremely popular, with many thousands of users around the world. Its functionality has grown through the efforts of many people. The time has come for a new version,  $\text{\LaTeX} 2\epsilon$ , which is described in this edition.  $\text{\LaTeX} 2\epsilon$  includes many of the enhancements that were made to  $\text{\LaTeX} 2.09$ , as well as some new ones.

I implemented most of  $\text{\LaTeX} 2.09$  myself.  $\text{\LaTeX} 2\epsilon$  was implemented by a group led by Frank Mittelbach, which included Johannes Braams, David Carlisle, Michael Downes, Alan Jeffrey, Chris Rowley, Sebastian Rahtz, and Rainer Schöpf. They were assisted by many testers of the new version, and by the following organizations: the American Mathematical Society, the Open University (UK), and the Zentrum für Datenverarbeitung der Universität Mainz. Lyle Ramshaw helped with the implementation of Bezier curves. My thanks to all of these people—especially Frank and Chris, with whom I have spent many enjoyable hours arguing about  $\text{\LaTeX}$ .

$\text{\LaTeX}$  has been made more useful by two programs:  $\text{BIBTeX}$ , written by Oren Patashnik, and  $\text{MakeIndex}$ , written by Pehong Chen and modified by Nelson Beebe.

Many people helped me write this book—often without knowing it. Advice given to me over the years by Cynthia Hibbard and Mary-Claire van Leunen has found its way onto a number of these pages. Andrei Broder was my local informant for Romanian. Helen Goldstein assisted with research on matters ranging from art to zoology.

This edition was improved by the corrections and suggestions of Marc Brown, Michel Goossens, and the implementers of  $\text{\LaTeX} 2\epsilon$ . Stephen Harrison helped produce the final output. Errors and infelicities in the first printing were found by Rosemary Bailey, Malcolm Clark, and Ellen Gilkerson. The following people found errors in, or suggested improvements to, the previous edition: Martín Abadi, Helmer Aslaksen, Barbara Beeton, Rick Clark, John DeTreville, Mathieu Federspiel, Michael Fischer, Stephen Gildea, Andy Hisgen, Joseph Hurler, Louis E. Janus, Dave Johnson, Charles Karney, Nori Kawasaki, Steve Kelem, Mark Kent, William LeFebvre, Jerry Leichter, Hank Lewis, Stephen Peckham,

Hal Perkins, Flavio Rose, Scott Simpson, David Sullivan, Matthew Swift, Walter Taylor, Joe Weening, Sam Whidden, Edgar Whipple, Chris Wilson, David Wise, and Rusty Wright. I also received helpful comments and complaints about preliminary versions of L<sup>A</sup>T<sub>E</sub>X and of the first edition of this book from Todd Allen, Robert Amsler, David Bacon, Stephen Barnard, Per Bothner, David Braunegg, Daniel Brotsky, Chuck Buckley, Pavel Curtis, Russell Greiner, Andrew Hanson, Michael Harrison, B. J. Herbison, Calvin W. Jackson, David Kosower, Kenneth Laws, Tim Morgan, Mark Moriconi, Stuart Reges, A. Wayne Slawson, David Smith, Michael Spivak, Mark Stickel, Gary Swanson, Mike Urban, Mark Wadsworth, and Gio Wiederhold. Assistance in the development of L<sup>A</sup>T<sub>E</sub>X 2.09 was provided by David Fuchs, Richard Furuta, Marshall Henrichs, Lynn Ruggles, Richard Southall, Chris Torek, Howard Trickey, and SRI International.

Since the introduction of version 2.09, my work on L<sup>A</sup>T<sub>E</sub>X has been supported by Digital Equipment Corporation. I want to thank Robert Taylor and all the other members of Digital's Systems Research Center for making it a fun place to work.

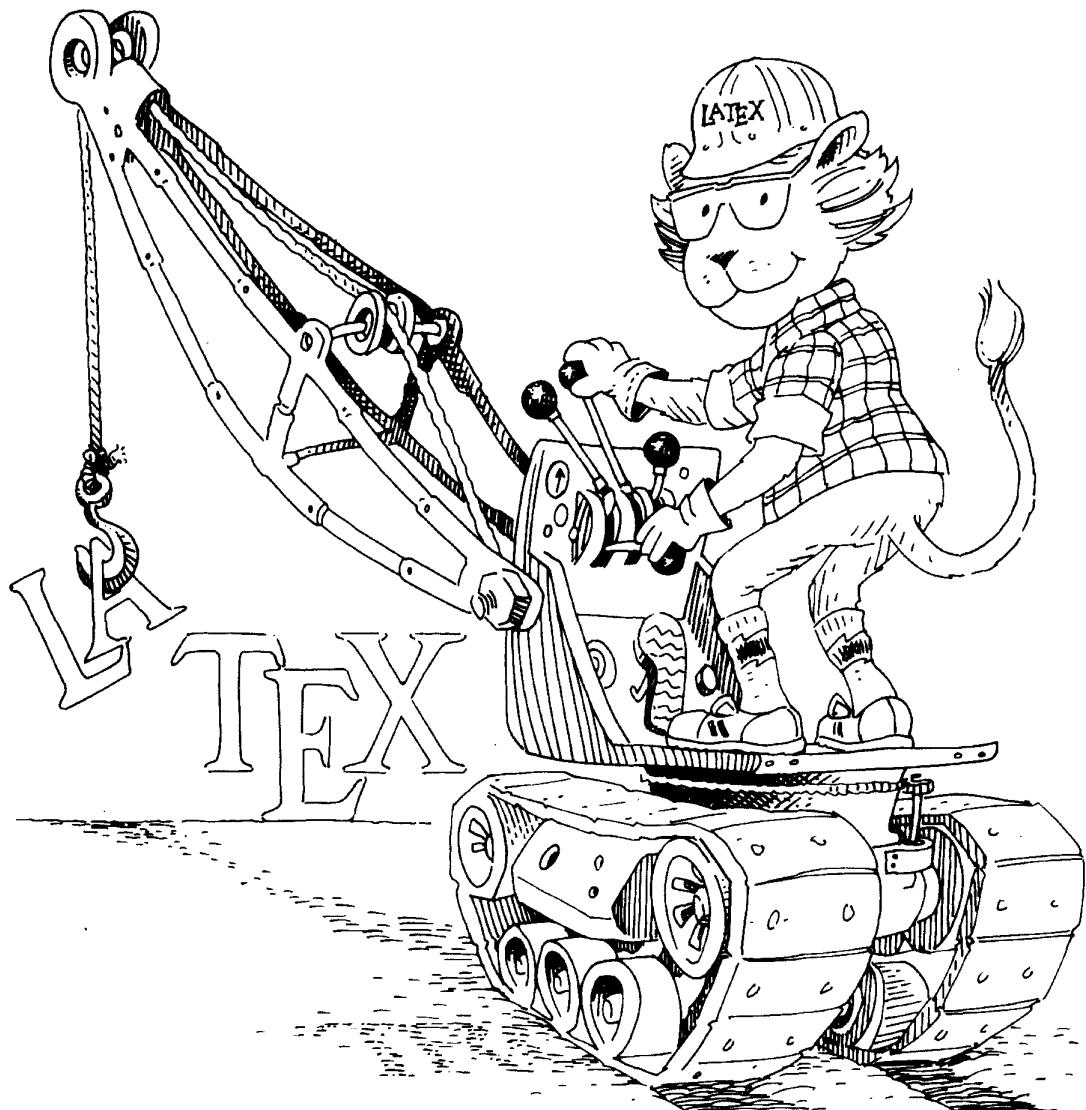
Finally, I want to express my special thanks to two men who made this book possible. Donald Knuth created T<sub>E</sub>X, the program on which L<sup>A</sup>T<sub>E</sub>X is based. He also answered all my questions, even the stupid ones, and was always willing to explain T<sub>E</sub>X's mysteries. Peter Gordon persuaded me to write the first edition, despite my doubts that anyone would buy a book about a typesetting system. Over the years, he has provided advice, fine dining, and friendship.

L. L.

Palo Alto, California  
September 1994

## CHAPTER 1

# Getting Acquainted



$\text{\LaTeX}$  is a system for typesetting documents. Its first widely available version, mysteriously numbered 2.09, appeared in 1985.  $\text{\LaTeX}$  is now extremely popular in the scientific and academic communities, and it is used extensively in industry. It has become a *lingua franca* of the scientific world; scientists send their papers electronically to colleagues around the world in the form of  $\text{\LaTeX}$  input.

Over the years, various nonstandard enhancements were made to  $\text{\LaTeX}$  2.09 to overcome some of its limitations.  $\text{\LaTeX}$  input that made use of these enhancements would not work properly at all sites. A new version of  $\text{\LaTeX}$  was needed to keep a Tower of Babel from rising. The current version of  $\text{\LaTeX}$ , with the somewhat less mysterious number 2 $\epsilon$ , was released in 1994.  $\text{\LaTeX}$  2 $\epsilon$  contains an improved method for handling different styles of type, commands for including graphics and producing colors, and many other new features.

Almost all standard  $\text{\LaTeX}$  2.09 input files will work with  $\text{\LaTeX}$  2 $\epsilon$ . However, to take advantage of the new features, users must learn a few new  $\text{\LaTeX}$  2 $\epsilon$  conventions.  $\text{\LaTeX}$  2.09 users should read Appendix D to find out what has changed. The rest of this book is about  $\text{\LaTeX}$ , which, until a newer version appears, means  $\text{\LaTeX}$  2 $\epsilon$ .

$\text{\LaTeX}$  is available for just about any computer made today. The versions that run on these different systems are essentially the same; an input file created according to the directions in this book should produce the same output with any of them. However, how you actually run  $\text{\LaTeX}$  depends upon the computer system. Moreover, some new features may not be available on all systems when  $\text{\LaTeX}$  2 $\epsilon$  is first released. For each computer system, there is a short companion to this book, titled something like *Local Guide to  $\text{\LaTeX}$  for the McKludge PC*, containing information specific to that system. I will call it simply the *Local Guide*. It is distributed with the  $\text{\LaTeX}$  software.

There is another companion to this book, *The  $\text{\LaTeX}$  Companion* by Goossens, Mittelbach, and Samarin [3]. This companion is an in-depth guide to  $\text{\LaTeX}$  and to its *packages*—standard enhancements that can be used at any site to provide additional features. The  *$\text{\LaTeX}$  Companion* is the place to look if you can't find what you need in this book. It describes more than a hundred packages.

## 1.1 How to Avoid Reading This Book

Many people would rather learn about a program at their computer than by reading a book. There is a small sample  $\text{\LaTeX}$  input file named `small2e.tex` that shows how to prepare your own input files for typesetting simple documents. Before reading any further, you might want to examine `small2e.tex` with a text editor and modify it to make an input file for a document of your own, then run  $\text{\LaTeX}$  on this file and see what it produces. The *Local Guide* will tell you how to find `small2e.tex` and run  $\text{\LaTeX}$  on your computer; it may also contain information about text editors. Be careful not to destroy the original version of `small2e.tex`; you'll probably want to look at it again.

The file `small2e.tex` is only forty lines long, and it shows how to produce only very simple documents. There is a longer file named `sample2e.tex` that contains more information. If `small2e.tex` doesn't tell you how to do something, you can try looking at `sample2e.tex`.

If you prefer to learn more about a program before you use it, read on. Almost everything in the sample input files is explained in the first two chapters of this book.

## 1.2 How to Read This Book

While `sample2e.tex` illustrates many of L<sup>A</sup>T<sub>E</sub>X's features, it is still only about two hundred lines long, and there is a lot that it doesn't explain. Eventually, you will want to typeset a document that requires more sophisticated formatting than you can obtain by imitating the two sample input files. You will then have to look in this book for the necessary information. You can read the section containing the information you need without having to read everything that precedes it. However, all the later chapters assume you have read Chapters 1 and 2. For example, suppose you want to set one paragraph of a document in small type. Looking up "type size" in the index or browsing through the table of contents will lead you to Section 6.7.1, which talks about "declarations" and their "scope"—simple concepts that are explained in Chapter 2. It will take just a minute or two to learn what to do if you've already read Chapter 2; it could be quite frustrating if you haven't. So, it's best to read the first two chapters now, before you need them.

L<sup>A</sup>T<sub>E</sub>X's input is a file containing the document's text together with commands that describe the document's structure; its output is a file of typesetting instructions. Another program must be run to convert these instructions into printed output. With a high-resolution printer, L<sup>A</sup>T<sub>E</sub>X can generate book-quality typesetting.

This book tells you how to prepare a L<sup>A</sup>T<sub>E</sub>X input file. The current chapter discusses the philosophy underlying L<sup>A</sup>T<sub>E</sub>X; here is a brief sketch of what's in the remaining chapters and appendices:

**Chapter 2** explains what you should know to handle most simple documents and to read the rest of the book. Section 2.5 contains a summary of everything in the chapter; it serves as a short reference manual.

**Chapter 3** describes logical structures for handling a variety of formatting problems. Section 3.4 explains how to define your own commands, which can save typing when you write the document and retying when you change it. It's a good idea to read the introduction—up to the beginning of Section 3.1—before reading any other part of the chapter.

**Chapter 4** contains features especially useful for large documents, including automatic cross-referencing and commands for splitting a large file into smaller pieces. Section 4.7 discusses sending your document electronically.

**Chapter 5** is about making books, slides, and letters (the kind you send by post).

**Chapter 6** describes the visual formatting of the text. It has information about changing the style of your document, explains how to correct bad line and page breaks, and tells how to do your own formatting of structures not explicitly handled by  $\text{\LaTeX}$ .

**Chapter 7** discusses pictures—drawing them yourself and inserting ones prepared with other programs—and color.

**Chapter 8** explains how to deal with errors. This is where you should look when  $\text{\LaTeX}$  prints an error message that you don't understand.

**Appendix A** describes how to use the *MakeIndex* program to make an index.

**Appendix B** describes how to make a bibliographic database for use with  $\text{BIBTeX}$ , a separate program that provides an automatic bibliography feature for  $\text{\LaTeX}$ .

**Appendix C** is a reference manual that compactly describes all  $\text{\LaTeX}$ 's features, including many advanced ones not described in the main text. If a command introduced in the earlier chapters seems to lack some necessary capabilities, check its description here to see if it has them. This appendix is a convenient place to refresh your memory of how something works.

**Appendix D** describes the differences between the current version of  $\text{\LaTeX}$  and the original version,  $\text{\LaTeX}$  2.09.

**Appendix E** is for the reader who knows  $\text{TeX}$ , the program on which  $\text{\LaTeX}$  is built, and wants to use  $\text{TeX}$  commands that are not described in this book.

When you face a formatting problem, the best place to look for a solution is in the table of contents. Browsing through it will give you a good idea of what  $\text{\LaTeX}$  has to offer. If the table of contents doesn't work, look in the index; I have tried to make it friendly and informative.

Each section of Chapters 3–7 is reasonably self-contained, assuming only that you have read Chapter 2. Where additional knowledge is required, explicit cross-references are given. Appendix C is also self-contained, but a command's description may be hard to understand without first reading the corresponding description in the earlier chapters.

The descriptions of most  $\text{\LaTeX}$  commands include examples of their use. In this book, examples are formatted in two columns, as follows:

The left column shows the printed output; the right column contains the input that produced it.

The left column shows the printed output; the right column contains the input that produced it.

Note the special typewriter type style in the right column. It indicates what you type—either text that you put in the input file or something like a file name that you type as part of a command to the computer.

Since the sample output is printed in a narrower column, and with smaller type, than  $\text{\LaTeX}$  normally uses, it won't look exactly like the output you'd get from that input. The convention of the output appearing to the left of the corresponding input is generally also used when commands and their output are listed in tables.

### 1.3 The Game of the Name

The  $\text{\TeX}$  in  $\text{\LaTeX}$  refers to Donald Knuth's  $\text{\TeX}$  typesetting system. The  $\text{\LaTeX}$  program is a special version of  $\text{\TeX}$  that understands  $\text{\LaTeX}$  commands. Think of  $\text{\LaTeX}$  as a house built with the lumber and nails provided by  $\text{\TeX}$ . You don't need lumber and nails to live in a house, but they are handy for adding an extra room. Most  $\text{\LaTeX}$  users never need to know any more about  $\text{\TeX}$  than they can learn from this book. However, the lower-level  $\text{\TeX}$  commands described in *The  $\text{\TeX}$ book* [4] can be very useful when creating a new package for  $\text{\LaTeX}$ .

I will use the term “ $\text{\TeX}$ ” when describing standard  $\text{\TeX}$  features and “ $\text{\LaTeX}$ ” when describing features unique to  $\text{\LaTeX}$ , but the distinction will be of interest mainly to readers already familiar with  $\text{\TeX}$ . You may ignore it and use the two names interchangeably.

One of the hardest things about using  $\text{\LaTeX}$  is deciding how to pronounce it. This is also one of the few things I'm not going to tell you about  $\text{\LaTeX}$ , since pronunciation is best determined by usage, not fiat.  $\text{\TeX}$  is usually pronounced *teck*, making *lah-teck*, *lah-teck*, and *lay-teck* the logical choices; but language is not always logical, so *lay-tecks* is also possible.

The written word carries more legal complications than the spoken, and the need to distinguish  $\text{\TeX}$  and  $\text{\LaTeX}$  from similarly spelled products restricts how you may write them. The best way to refer to these programs is by their logos, which can be generated with simple  $\text{\LaTeX}$  commands. When this is impossible, as in an e-mail message, you should write them as  $\text{\TeX}$  and  $\text{\LaTeX}$ , where the unusual capitalization identifies these computer programs.

### 1.4 Turning Typing into Typography

Traditionally, an author provides a publisher with a typed manuscript. The publisher's typographic designer decides how the manuscript is to be formatted, specifying the length of the printed line, what style of type to use, how much

space to leave above and below section headings, and many other things that determine the printed document's appearance. The designer writes a series of instructions to the typesetter, who uses them to decide where on the page to put each of the author's words and symbols. In the old days, typesetters produced a matrix of metal type for each page; today they produce computer files. In either case, their output is used to control the machine that does the actual typesetting.

$\text{\LaTeX}$  is your typographic designer, and  $\text{\TeX}$  is its typesetter. The  $\text{\LaTeX}$  commands that you type are translated into lower-level  $\text{\TeX}$  typesetting commands. Being a modern typesetter,  $\text{\TeX}$  produces a computer file, called the *device-independent* or *dvi* file. The *Local Guide* explains how to use this file to generate a printed document with your computer. It also explains how to view your document on your computer, using a screen previewer. Unless your document is very short, you will want to see the typeset version as you're writing it. Use a previewer instead of laying waste to our planet's dwindling forests by printing lots of intermediate versions. In fact, unless you want to take a copy with you on a wilderness expedition, you may never have to print it at all. It is easier and faster to distribute your document electronically than by mailing paper copies.

A human typographic designer knows what the manuscript is generally about and uses this knowledge in deciding how to format it. Consider the following typewritten manuscript:

```
The German mathematician Kronecker, sitting
quietly at his desk, wrote:
 God created the whole numbers; all
 the rest is man's work.
Seated in front of the terminal, with Basic
hanging on my every keystroke, I typed:
 for i = 1 to infinity
 let number[i] = i
```

A human designer knows that the first indented paragraph (God created ...) is a quotation and the second is a computer program, so the two should be formatted differently. He would probably set the quotation in ordinary roman type and the computer program in a typewriter type style.  $\text{\LaTeX}$  is only a computer program and can't understand English, so it can't figure all this out by itself. It needs more help from you than a human designer would.

The function of typographic design is to help the reader understand the author's ideas. For a document to be easy to read, its visual structure must reflect its logical structure. Quotations and computer programs, being logically distinct structural elements, should be distinguished visually from one another. The designer should therefore understand the document's logical structure. Since  $\text{\LaTeX}$  can't understand your prose, you must explicitly indicate the logical structure by typing special commands. The primary function of almost all the  $\text{\LaTeX}$

commands that you type should be to describe the logical structure of your document. As you are writing your document, you should be concerned with its logical structure, not its visual appearance. The L<sup>A</sup>T<sub>E</sub>X approach to typesetting can therefore be characterized as *logical design*.

## 1.5 Why L<sup>A</sup>T<sub>E</sub>X?

When L<sup>A</sup>T<sub>E</sub>X was introduced in 1985, few authors had the facilities for typesetting their own documents. Today, desktop publishing is commonplace. You can buy a “WYSIWYG” (what you see is what you get) program that lets you see exactly what your document will look like as you type it. WYSIWYG programs are very appealing. They make it easy to put text wherever you want in whatever size and style of type you want. Why use L<sup>A</sup>T<sub>E</sub>X, which requires you to tell it that a piece of text is a quotation or a computer program, when a WYSIWYG program allows you to format the text just the way you want it?

WYSIWYG programs replace L<sup>A</sup>T<sub>E</sub>X’s logical design with *visual design*. Visual design is fine for short, simple documents like letters and memos. It is not good for more complex documents such as scientific papers. WYSIWYG has been characterized as “what you see is all you’ve got”.<sup>1</sup> To illustrate the advantage of logical over visual design, I will consider a simple example from the file `sample2e.tex`.

Near the top of the second page of the document is the mathematical term  $(A, B)$ . With a WYSIWYG program, this term is entered by typing  $(A, B)$ . You could type it the same way in the L<sup>A</sup>T<sub>E</sub>X input. However, the term represents a mathematical structure—the inner product of  $A$  and  $B$ . An experienced L<sup>A</sup>T<sub>E</sub>X user will define a command to express this structure. The file `sample2e.tex` defines the command `\ip` so that `\ip{A}{B}` produces  $(A, B)$ . The term  $(\Gamma, \psi')$  near the end of the document is also an inner product and is produced with the `\ip` command.

Suppose you decide that there should be a little more space after the comma in an inner product. Just changing the definition of the `\ip` command will change  $(A, B)$  to  $(A, B)$  and  $(\Gamma, \psi')$  to  $(\Gamma, \psi')$ . With a WYSIWYG program, you would have to insert the space by hand in each formula—not a problem for a short document with two such terms, but a mathematical paper could contain dozens and a book could contain hundreds. You would probably produce inconsistent formatting by missing some formulas or forgetting to add the space when entering new ones. With L<sup>A</sup>T<sub>E</sub>X, you don’t have to worry about formatting while writing your document. Formatting decisions can be made and changed at any time.

The advantage of logical design becomes even more obvious if you decide that you prefer the notation  $\langle A|B \rangle$  for the inner product of  $A$  and  $B$ . The

---

<sup>1</sup>Brian Reid attributes this phrase to himself and/or Brian Kernighan.

file `sample2e.tex` contains an alternate definition of `\ip` that produces this notation.

Typing `\ip{A}{B}` is just a little more work than typing `(A,B)` (though it's a lot easier than entering `\langle A|B \rangle` if the symbols “⟨” and “⟩” must be chosen with a mouse from a pull-down menu). But this small effort is rewarded by the benefits of maintaining the logical structure of your document instead of just its visual appearance.

One advantage of WYSIWYG programs is that you can see the formatted version of your document while writing it. Writing requires reading what you have already written. Although you want L<sup>A</sup>T<sub>E</sub>X to know that the term is an inner product, you would like to read `(A,B)` or `\langle A|B \rangle`, not `\ip{A}{B}`. The speed of modern computers has eliminated much of this advantage. I now type a couple of keystrokes and, a few seconds later, a typeset version of the section I am working on appears on my screen. As computers get faster, those few seconds will turn into a fraction of a second.

## 1.6 Turning Ideas into Input

The purpose of writing is to present your ideas to the reader. This should always be your primary concern. It is easy to become so engrossed with form that you neglect content. Formatting is no substitute for writing. Good ideas couched in good prose will be read and understood, regardless of how badly the document is formatted. L<sup>A</sup>T<sub>E</sub>X was designed to free you from formatting concerns, allowing you to concentrate on writing. If you spend a lot of time worrying about form, you are misusing L<sup>A</sup>T<sub>E</sub>X.

Even if your ideas are good, you can probably learn to express them better. The classic introduction to writing English prose is Strunk and White's brief *Elements of Style* [6]. A more complete guide to using language properly is Theodore Bernstein's *The Careful Writer* [1]. These two books discuss general writing style. Writers of scholarly or technical prose need additional information. Mary-Claire van Leunen's *Handbook for Scholars* [7] is a delightful guide to academic and scholarly writing. The booklet titled *How to Write Mathematics* [5] can help scientists and engineers as well as mathematicians. It's also useful to have a weightier reference book at hand; *Words into Type* [8] and the *Chicago Manual of Style* [2] are two good ones.

## 1.7 Trying It Out

You may already have run L<sup>A</sup>T<sub>E</sub>X with input based on the sample files. If not, this is a good time to learn how. The section in the *Local Guide* titled *Running a Sample File* explains how to obtain a copy of the file `sample2e.tex` and run L<sup>A</sup>T<sub>E</sub>X with it as input. Follow the directions and see what L<sup>A</sup>T<sub>E</sub>X can do.

After printing the document generated in this way, try changing the document's format. Using a text editor, examine the file `sample2e.tex`. A few lines down from the beginning of the file is a line that reads

```
\documentclass{article}
```

Change that line to

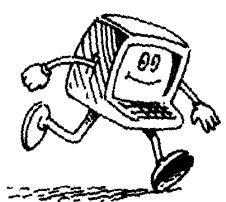
```
\documentclass[twocolumn]{article}
```

Save the changed file under the name `chgsam.tex`, and use this file to print a new version of the document. To generate the new version, do exactly what you did the last time, except type `chgsam` wherever you had typed `sample2e`. Comparing the two printed versions shows how radically the appearance of the document can be altered by a simple change to a command. To try still another format, change `chgsam.tex` so the line above reads

```
\documentclass[12pt]{article}
```

and use the changed file to print a third version of the document.

From now on, I will usually ignore the process of going from the  $\text{\LaTeX}$  input file to the printed output and will write something like: "Typing --- produces a long dash." What this really means is that putting the three characters --- in your input file will, when that file is processed by  $\text{\LaTeX}$  and the device-independent file printed, produce a long dash in the printed output.



## CHAPTER 2

# Getting Started



## 2.1 Preparing an Input File

The input to  $\text{\LaTeX}$  is a text file. I assume that you know how to use a text editor to create such a file, so I will tell you only what should go into your input file, not how to get it there. A good text editor can be customized by the user to make it easier to prepare  $\text{\LaTeX}$  input files. Consult the *Local Guide* to find out how to customize the text editors on your computer.

On most computers, file names have two parts separated by a period, like `sample2e.tex`. I will call the first part its *first name* and the second part its *extension*, so `sample2e` is the first name of `sample2e.tex`, and `tex` is its extension. Your input file's first name can be any name allowed by your computer system, but its extension should be `tex`.

Let's examine the characters that can appear in your input file. First, there are the upper- and lowercase letters and the ten digits 0 ... 9. Don't confuse the uppercase letter `O` (oh) with the digit 0 (zero), or the letter `l` (the lowercase *el*) with the digit 1 (one). Next, there are the following sixteen punctuation characters:

. : ; , ? ! ' ' ( ) [ ] - / \* @

Note that there are two different quote symbols: ' and '. You may think of ' as an ordinary "single quote" and ' as a funny symbol, perhaps displayed like ` on your screen. The *Local Guide* should tell where to find ' and ' on your keyboard, if they're not obvious. The characters ( and ) are ordinary parentheses, while [ and ] are called *square brackets*, or simply *brackets*.

The ten special characters

# \$ % & ^ \_ \ { }

are used only in  $\text{\LaTeX}$  commands. Check the *Local Guide* for help in finding them on your keyboard. The character \ is called *backslash*, and should not be confused with the more familiar /, as in 1/2. Most  $\text{\LaTeX}$  commands begin with a \ character, so you will soon become very familiar with it. The { and } characters are called *curly braces* or simply *braces*.

The five characters

+ = | < >

are used mainly in mathematical formulas, although + and = can be used in ordinary text. The character " (double quote) is hardly ever used.

Unless your *Local Guide* tells you otherwise, these are the only characters that you should see when you look at a  $\text{\LaTeX}$  input file. However, there are other "invisible" characters in your file: space characters, such as the one you usually enter by pressing the *space bar*, and special characters that indicate the end of a line, usually entered by pressing the *return* key (sometimes labeled *enter*). These invisible characters are all considered the same by  $\text{\TeX}$ , and I will

treat them as if they were a single character called *space*, which I will sometimes denote by `\s`. Any sequence of space characters is handled the same as a single one, so it doesn't matter if the space between two words is formed by one space character or several of them. However, a blank line—one containing nothing but space characters—is interpreted by TeX as the end of a paragraph. Some text editors organize a file into pages. TeX acts as if there were a blank line between the pages of such a file.

## 2.2 The Input

Most L<sup>A</sup>T<sub>E</sub>X commands describe the logical structure of the document. To understand these commands, you must know how L<sup>A</sup>T<sub>E</sub>X perceives that logical structure. A document contains logical structures of different sizes, from chapters down through individual letters. We start by considering the very familiar intermediate-sized structures: sentences and paragraphs.

### 2.2.1 Sentences and Paragraphs

Describing simple sentences and paragraphs to L<sup>A</sup>T<sub>E</sub>X poses no problem; you pretty much type what comes naturally.

The ends of words and sentences are marked by spaces. It doesn't matter how many spaces you type; one is as good as 100.

One or more blank lines denote the end of a paragraph.

The ends of words and sentences are marked by spaces. It doesn't matter how many spaces you type; one is as good as 100.

One or more blank lines denote the end of a paragraph.

TeX ignores the way the input is formatted, paying attention only to the logical concepts end-of-word, end-of-sentence, and end-of-paragraph.

That's all you have to know for typing most of your text. The remainder of this book is about how to type the rest, starting with some other things that occur fairly frequently in ordinary sentences and paragraphs.

### Quotation Marks

Typewritten text uses only two quotation-mark symbols: a double quote " and single quote ', the latter serving also as an apostrophe. Printed text, however, uses a left and a right version of each, making four different symbols. TeX interprets the character ' as a single left quote, and the character ' as a single right quote. To get a double quote, just type two single quotes.

'Convention' dictates that punctuation go inside quotes, like "this," but I think it's better to do "this".

'Convention' dictates that punctuation go inside quotes, like "this," but I think it's better to do "this".

Remember that the right-quote character ' is the one you're used to thinking of as a single quote, and the left-quote character ' is the one you're probably unfamiliar with. An apostrophe is produced with the usual ' character.

Typing a double quote followed by a single quote, or vice versa, poses a problem because something like " " would be ambiguous. The solution is to type the command \, (a \ character followed by a comma) between the two quotation marks.

"'Fi' or 'fum?'" he asked.

"\,'Fi' or 'fum?'\,," he asked.

The \, is a typesetting command that causes *T<sub>E</sub>X* to insert a small amount of space. Don't leave any space in the input file before or after the \, command.

### Dashes

You can produce three different sizes of dash by typing one, two, or three "—" characters:

An intra-word dash or hyphen, as in X-ray.  
A medium dash for number ranges, like 1-2.  
A punctuation dash—like this.

An intra-word dash or hyphen, as in X-ray.  
A medium dash for number ranges, like 1--2.  
A punctuation dash---like this.

There is usually no space before or after a dash. Minus signs are not dashes; they should appear only in mathematical formulas, which are discussed below.

### Space After a Period

Typesetters often put a little extra space after a sentence-ending period. This is easy for a human typesetter, but not for a program like *T<sub>E</sub>X* that has trouble deciding which periods end sentences. Instead of trying to be clever, *T<sub>E</sub>X* simply assumes that a period ends a sentence unless it follows an uppercase letter. This works most of the time, but not always—abbreviations like "etc." being the most common exception. You tell *T<sub>E</sub>X* that a period doesn't end a sentence by using a \\_ command (a \ character followed by a space or the end of a line) to make the space after the period.

Tinker et al. made the double play.

Tinker et al.\ made the double play.

It doesn't matter how many spaces you leave after the \ character, but don't leave any space between the period and the backslash. The \\_ command produces an ordinary interword space, which can also be useful in other situations.

On the rare occasions that a sentence-ending period follows an uppercase letter, you will have to tell *T<sub>E</sub>X* that the period ends the sentence. You do this by preceding the period with a \@ command.

The Romans wrote I + I = II. Really!

The Romans wrote I + I = II\@. Really!

If a sentence-ending period is followed by a right parenthesis or a right quote (single or double), then the period's extra space goes after the parenthesis or quote. In these cases too,  $\text{\TeX}$  will need a hand if its assumption that a period ends a sentence unless it follows an uppercase letter is wrong.

“Beans (lima, etc.) have vitamin B.”

‘‘Beans (lima, etc.)\ have vitamin B\@.’’

Extra space is also added after a question mark (?), exclamation point (!), or colon (:) just as for a period—that is, unless it follows an uppercase letter. The  $\backslash_!$  and  $\backslash@$  commands are used the same way with each of these punctuation characters.

### Special Symbols

Remember those ten special characters, mentioned on page 12, that you type only as part of  $\text{\LaTeX}$  commands? Some of them, like \$, represent symbols that you might very well want in your document. Seven of those symbols can be produced by typing a \ in front of the corresponding character.

$\$$  &  $\%$   $\#$  - { } are easy to produce.

$\backslash$$   $\backslash\&$   $\backslash\%$   $\backslash\#$   $\backslash\_$   $\backslash\{$   $\backslash\}$  are easy to produce.

The other three special characters ( $\sim$ ,  $\hat{}$ , and  $\backslash$ ) usually appear only in simulated keyboard input, which is produced with the commands described in Section 3.7.

You can get  $\text{\LaTeX}$  to produce any symbol that you're likely to want, and many more besides, such as:  $\S$   $\mathcal{L}$   $\psi$   $\star$   $\otimes$   $\approx$   $\bowtie$   $\Leftarrow$   $\Leftarrowtail$   $\clubsuit$ . Sections 3.2 and 3.3.2 tell how.

### Simple Text-Generating Commands

Part of a sentence may be produced by a text-generating command. For example, the  $\text{\TeX}$  and  $\text{\LaTeX}$  logos are produced by the commands  $\backslash\text{\TeX}$  and  $\backslash\text{\LaTeX}$ , respectively.

Some people use plain  $\text{\TeX}$ , but I prefer  $\text{\LaTeX}$ .

Some people use plain  $\backslash\text{\TeX}$ , but I prefer  $\backslash\text{\LaTeX}$ .

A useful text-generating command is  $\backslash\text{today}$ , which produces the current date.

This page was produced on May 18, 1994.

This page was produced on  $\backslash\text{today}$ .

Another useful text-generating command is  $\backslash\text{ldots}$ , which produces an *ellipsis*—the sequence of three dots used to denote omitted material. (Simply typing three periods doesn't produce the right spacing between the dots.)

If nominated ..., I will not serve.

If nominated  $\backslash\text{ldots}$ , I will not serve.

Most of the command names you've seen so far have consisted of a `\` (backslash) followed by a single nonletter. From now on, most commands you will encounter have names consisting of a `\` followed by one or more letters. In reading the input file, `TeX` knows it has come to the end of such a command name when it finds a nonletter: a digit like "7", a punctuation character like ";", a special character like "\", a space, or the end of a line. The most common way to end this kind of command name is with a space or end of line, so `TeX` ignores all spaces following it. If you want a space after the logo produced by the `\LaTeX` command, you can't just leave a space after the command name; all such spaces are ignored. You must tell `TeX` to put in the space by typing a `\_` command.

This page of the `LATEX` manual was produced on  
May 18, 1994.

This page of the `\LaTeX\` manual was  
produced on `\today`.

Note how `TeX` ignored the space after the `\today` command in the input and did not produce any space after the date in the output.

The case of letters matters in a command name; typing `\Today` produces an error, because the correct command name is `\today`. Most command names have only lowercase letters.

### Emphasizing Text

Emphasized text is usually underlined in a typewritten manuscript and *italicized* in a printed document. Underlining and italics are visual concepts; when typing your document, you should be concerned only with the logical concept of emphasis. The `\emph` command tells `LATEX` that text is to be emphasized.

Here is some silly *emphasized text*.

Here is some silly `\emph{emphasized text}`.

In the `\emph{emphasized text}` command, `\emph` is the command name and `emphasized text` is its *argument*. Most commands have either no arguments, like `\today`, or a single argument, like `\emph`. However, there are a few with multiple arguments, each of which is enclosed in braces. Spaces are ignored between the arguments, and between the command name and its first argument.

Commands like `\emph` can be nested within one another in the obvious way. Most styles use ordinary roman type for emphasized text that appears inside emphasized text.

You can have *emphasized text* within *emphasized text* too.

You can have `\emph{emphasized text}`  
`\emph{within}` `\emph{emphasized text}` too.

Emphasis should be used sparingly. Like raising your voice, it is an effective way to get attention, but an annoying distraction if done too often.

## Preventing Line Breaks

In putting text onto paper, a paragraph must be broken into lines of print. Text becomes hard to read if a single logical unit is split across lines in an arbitrary fashion, so typesetters break lines between words when possible and split words only between syllables (inserting a hyphen at the break). Sometimes a line should not be broken between or within certain words. Human typesetters recognize these situations, but  $\text{\TeX}$  must be told about some of them.

Line breaking should be prevented at certain interword spaces. For example, the expression “Chapter 3” looks strange if the “Chapter” ends one line and the “3” begins the next. Typing  $\sim$  (a tilde character) produces an ordinary interword space at which  $\text{\TeX}$  will never break a line. Below are some examples indicating when a  $\sim$  should be used.

|                           |                     |                  |
|---------------------------|---------------------|------------------|
| Mr. $\sim$ Jones          | Figure $\sim$ 7     | (1) $\sim$ gnats |
| U. $\sim$ S. $\sim$ Grant | from 1 to $\sim$ 10 |                  |

It is best not to break a line within certain words. For example, you should try to avoid splitting a name (especially your own). The  $\text{\mbox}$  command tells  $\text{\TeX}$  to print its entire argument on the same line. In the following example,  $\text{\TeX}$  will never split “Lamport” across lines.

Doctor Lamport, I presume?

Doctor  $\text{\mbox}{\{Lamport\}}$ , I presume?

Most line breaks separate logically related units, and it would be nice if they could be avoided. However, unless you print your document on a mile-long strip of paper tape, line breaking is a necessary evil. Using too many  $\sim$  and  $\text{\mbox}$  commands leaves too few places to break lines. Inhibit line breaking only where necessary.

## Footnotes

Footnotes are produced with a  $\text{\footnote}$  command having the text of the footnote as its argument.

Gnus<sup>1</sup> can be quite a gnusance.

⋮

Gnus $\text{\footnote}{\{A gnu is a big animal.\}}$  can be quite a gnusance.

<sup>1</sup>A gnu is a big animal.

There is no space between the Gnus and the  $\text{\footnote}$  in this example; adding space would have put an unwanted space between the text and the footnote marker (the <sup>1</sup>).

A  $\text{\footnote}$  command cannot be used in the argument of most commands; for example, it can't appear in the argument of an  $\text{\mbox}$  command. Section C.3.3 explains how to footnote text that appears in a command argument.

## Formulas

If you're writing a technical document, it's likely to contain mathematical formulas. A formula appearing in the middle of a sentence is enclosed by `\(` and `\)` commands.

The formula  $x - 3y = 7$  is easy to type.

The formula `\( x-3y = 7 \)` is easy to type.

Any spaces that you type in the formula are ignored.

Does  $x + y$  always equal  $y + x$ ?

Does `\( x + y \)` always equal `\( y+x \)`?

TeX regards a formula as a word, which may be broken across lines at certain points, and space before the `\(` or after the `\)` is treated as an ordinary interword separation.

Subscripts are produced by the `_` command and superscripts by the `^` command.

$a_1 > x^{2n}/y^{2n}$

`\( a_{1} > x^{2n} / y^{2n} \)`

These two commands can be used only inside a mathematical formula.

When used in a formula, the right-quote character `'` produces a prime  $(')$ , two in a row produce a double prime, and so on.

This proves that  $x' < x'' - y'_3 < 10x'''z$ .

$\dots \quad \backslash( x' < x'' - y'_3 < 10 x''' z \backslash)$ .

Mathematical formulas can get very complex; Section 3.3 describes many additional commands for producing them. Here, I consider the use of formulas in the text. A formula is a grammatical unit; it should be treated as such in the sentence structure.

The formula  $a < 7$  is a noun in this sentence. It is sometimes used as a clause by writing that  $a < 7$ .

The formula `\( a < 7 \)` is a noun in this sentence. It is sometimes used ...

Beginning a sentence with a formula makes it hard to find the start of the sentence; don't do it. It is best to use a formula as a noun; it should certainly never appear as a complete sentence in the running text.

A variable like  $x$  is a formula. To save you some typing, L<sup>A</sup>T<sub>E</sub>X treats `$...$` the same as `\(...\)`.

Let  $x$  be a prime such that  $y > 2x$ .

Let `$x$` be a prime such that `$y>2x$`.

Use `$...$` only for a short formula, such as a single variable. It's easy to forget one of the `$` characters that surrounds a long formula. You can also type

`\begin{math} \dots \end{math}`

instead of `\(...\)`. You might want to use this form for very long formulas.

## Ignorable Input

When  $\text{\TeX}$  encounters a  $\%$  character in the input, it ignores the  $\%$  and all characters following it on that line—including the space character that ends the line.  $\text{\TeX}$  also ignores spaces at the beginning of the next line.

Gnus and armadillos are generally tolerant of one another and seldom quarrel.  $\text{Gnus and armadi\% More @_#!$^{\&} gnus? llos are generally ...}$

The  $\%$  has two uses: ending a line without producing any space in the output<sup>1</sup> and putting a comment (a note to yourself) in the input file.

## 2.2.2 The Document

We now jump from the document's intermediate-sized logical units to its largest one: the entire document itself. The text of every document starts with a `\begin{document}` command and ends with an `\end{document}` command.  $\text{\LaTeX}$  ignores anything that follows the `\end{document}`. The part of the input file preceding the `\begin{document}` command is called the *preamble*.

### The Document Class

The preamble begins<sup>2</sup> with a `\documentclass` command whose argument is one of the predefined classes of document that  $\text{\LaTeX}$  knows about. The file `sample2e.tex` begins with

```
\documentclass{article}
```

which selects the `article` class. The other standard  $\text{\LaTeX}$  class used for ordinary documents is the `report` class. The `article` class is generally used for shorter documents than the `report` class. Other standard classes are described in Chapter 5.

In addition to choosing the class, you can also select from among certain document-class options. The options for the `article` and `report` classes include the following:

`11pt` Specifies a size of type known as *eleven point*, which is ten percent larger than the ten-point type normally used.

`12pt` Specifies a twelve-point type size, which is twenty percent larger than ten point.

`twoside` Formats the output for printing on both sides of the page. ( $\text{\LaTeX}$  has no control over the actual printing.)

---

<sup>1</sup>However, you can't split a command name across two lines.

<sup>2</sup>As explained in Section 4.7, the `\documentclass` command may actually be preceded by prepended files.

`twocolumn` Produces two-column output.

Other options are described elsewhere in this book; all the standard ones are listed in Section C.5.1. Your *Local Guide* tells what others are available on your computer.

You specify a document-class option by enclosing it in square brackets immediately after the “`\documentclass`”, as in

```
\documentclass [twoside]{report}
```

Multiple options are separated by commas.

```
\documentclass [twocolumn,12pt]{article}
```

Don’t leave any space inside the square brackets.

The `\documentclass` command can be used either with or without the option-choosing part. The options, enclosed in square brackets, are an *optional argument* of the command. It is a L<sup>A</sup>T<sub>E</sub>X convention that optional arguments are enclosed in square brackets, while mandatory arguments are enclosed in curly braces. T<sub>E</sub>X ignores spaces after a command name like `\documentclass` and between command arguments.

The document class defines the commands for specifying L<sup>A</sup>T<sub>E</sub>X’s standard logical structures. Additional structures are defined by *packages*, which are loaded by the `\usepackage` command. For example, the command

```
\usepackage{latexsym}
```

loads the `latexsym` package, which defines commands to produce certain special math symbols. (See Section 3.3.2.) A package can have options, specified by an optional argument of `\usepackage` just like the one for `\documentclass`.

You will probably want to define some new commands for the special structures used in your particular document. For example, if you’re writing a cookbook you will probably define your own commands for formatting recipes, as explained in Section 3.4. These definitions should go in the preamble, after the `\documentclass` and `\usepackage` commands. The preamble can also contain commands to change some aspects of the formatting. If you have commands or format changes that you use in several documents, you may want to define your own package, as described in Section 6.1.4.

### The Title “Page”

A document usually has a title “page” listing its title, one or more authors, and a date. I write “page” in quotes because, for a short document, this information may be listed on the first page of text rather than on its own page. The title information consists of the title itself, the author(s), and the date; it is specified by the three declarations `\title`, `\author`, and `\date`. The actual title “page” is generated by a `\maketitle` command.

```

Gnus of the World
\title{Gnus of the World}
R. Dather J. Pennings B. Talkmore
\author{R. Dather \and J. Pennings
\and B. Talkmore}
4 July 1997
\date{4 July 1997}
...
\maketitle

```

Note how multiple authors are separated by `\and` commands.

The `\maketitle` command comes after the `\begin{document}`, usually before any other text. The `\title`, `\author`, and `\date` commands can come anywhere before the `\maketitle`. The `\date` is optional; L<sup>A</sup>T<sub>E</sub>X supplies the current date if the declaration is omitted, but the `\title` and `\author` must appear if a `\maketitle` command is used. Commands for adding other information, such as the author's address and an acknowledgment of support, are described in Section C.5.4.

### 2.2.3 Sectioning

Sentences are organized into paragraphs, and paragraphs are in turn organized into a hierarchical *section structure*. You are currently reading Subsection 2.2.3, titled *Sectioning*, which is part of Section 2.2, titled *The Input*, which in turn is part of Chapter 2, titled *Getting Started*. I will use the term *sectional units* for things like chapters, sections, and subsections.

A sectional unit is begun by a sectioning command with the unit's title as its argument.

#### 4.7 A Sectioning Command

L<sup>A</sup>T<sub>E</sub>X automatically generates the section number. Blank lines before or after a sectioning command have no effect.

```
\subsection{A Sectioning Command}
```

```
\LaTeX\ automatically generates the section
number. Blank lines before or after a ...
```

The document class determines what sectioning commands are provided, the standard classes have the following ones:<sup>3</sup>

|                       |                             |                            |
|-----------------------|-----------------------------|----------------------------|
| <code>\part</code>    | <code>\subsection</code>    | <code>\paragraph</code>    |
| <code>\chapter</code> | <code>\subsubsection</code> | <code>\subparagraph</code> |
| <code>\section</code> |                             |                            |

The `article` document class does not contain the `\chapter` command, which makes it easy to include an “article” as a chapter of a “report”. The example above, like most others in this book, assumes the `article` document class, the 4.7 indicating that this is the seventh subsection of Section 4. In the `report` class, this subsection might be numbered 5.4.7, with “5” being the chapter number.

---

<sup>3</sup>The names `\paragraph` and `\subparagraph` are unfortunate, since they denote units that are often composed of several paragraphs; they have been retained for historical reasons.

The sectional unit denoted by each of these commands must appear as a subunit of the one denoted by the preceding command, except that the use of `\part` is optional. A subsection must be part of a section, which, in the `report` class, must be part of a chapter.

The `\part` command is used for major divisions of long documents; it does not affect the numbering of smaller units—in the `article` class, if the last section of Part 1 is Section 5, then the first section of Part 2 is Section 6.

If there is an appendix, it is begun with an `\appendix` command and uses the same sectioning commands as the main part of the document. The `\appendix` command does not produce any text; it simply causes sectional units to be numbered properly for an appendix.

The document class determines the appearance of the section title, including whether or not it is numbered. Declarations to control section numbering are described in Section C.4, which also tells you how to make a table of contents.

The argument of a sectioning command may be used for more than just producing the section title; it can generate a table of contents entry and a running head at the top of the page. (Running heads are discussed in Section 6.1.2.) When carried from where it appears in the input file to the other places it is used, the argument of a sectioning command is shaken up quite a bit. Some L<sup>A</sup>T<sub>E</sub>X commands are *fragile* and can break when they appear in an argument that is shaken in this way. Fragile commands are rarely used in the argument of a sectioning command. Of the commands introduced so far, the only fragile ones are `\(`, `\)`, `\begin`, `\end`, and `\footnote`—none of which you’re likely to need in a section title.<sup>4</sup> On the rare occasions when you have to put a fragile command in a section title, you simply protect it with a `\protect` command. The `\protect` command goes right before every fragile command’s name, as in:

```
\subsection {Is \protect\(\ x+y \protect\)\ Prime?}
```

This is actually a silly example because `$` is not a fragile command, so you can instead type

```
\subsection {Is $x + y$ Prime?}
```

but, because the problem is so rare, it’s hard to find a good example using the commands described in this chapter.

An argument in which fragile commands need `\protect` will be called a *moving* argument. Commands that are not fragile will be called *robust*. For any command that one might reasonably expect to use in a moving argument, I will indicate whether it is robust or fragile. Except in special cases mentioned in Chapter 6 and Appendix C, a `\protect` command can’t hurt, so it is almost always safe to use one when you’re not sure if it’s necessary.

---

<sup>4</sup>Section C.3.3 tells you how to footnote a section title.

### 2.2.4 Displayed Material

We return now to the level of the individual sentence. A sentence like

He turned and said to me, "My answer is no!", and then he left.

contains a complete sentence quoted within it. An entire paragraph can even appear inside a sentence, as in

He turned and said to me: "I've done all I'm going to. I refuse to have any further part in it. My answer is no!", and then he left.

It's hard to understand this sentence the way it is written. However, there's no problem if you read it aloud using a different tone of voice for the quotation. The typographic analog of changing your tone of voice is setting text off by indentation, also called *displaying*. The sentence above is much easier to read when typeset as follows:

He turned and said to me:

I've done all I'm going to. I refuse to have any further part  
in it. My answer is no!

and then he left.

Displayed material functions logically as a lower-level unit than a sentence, though grammatically it may consist of part of a sentence, a whole sentence, or even several paragraphs. To decide whether a portion of text should be a display or a separate sectional unit, you must determine if it is logically subordinate to the surrounding text or functions as an equal unit.

Quotations are often displayed.

The following is an example of a short displayed quotation.

... it's a good idea to make your input  
file as easy to read as possible.

It is indented at both margins.

... example of a short displayed quotation.  
\begin{quote}  
 \ldots\ it's a good idea to make your  
 input file as easy to read as possible.  
\end{quote}  
It is indented at both margins.

This example illustrates a type of *L<sup>A</sup>T<sub>E</sub>X* construction called an *environment*, which is typed

`\begin{name} ... \end{name}`

where *name* is the name of the environment. The *quote* environment produces a display suitable for a short quotation. You've already encountered two other examples of environments: the *math* environment and the *document* environment.

The standard *L<sup>A</sup>T<sub>E</sub>X* document classes provide environments for producing several different kinds of displays. Blank lines before or after the environment

mark a new paragraph. Thus, a blank line after the `\end` command means that the following text starts a new paragraph. Blank lines before and after the environment mean that it is a complete paragraph. It's a bad idea to start a paragraph with displayed material, so you should not have a blank line before a display environment without a blank line after it. Blank lines immediately following a display environment's `\begin` command and immediately preceding its `\end` command are ignored.

## Quotations

**L<sup>A</sup>T<sub>E</sub>X** provides two different environments for displaying quotations. The `quote` environment is used for either a short quotation or a sequence of short quotations separated by blank lines.

Our presidents have been known for their pithy remarks.

The buck stops here. *Harry Truman*  
 I am not a crook. *Richard Nixon*  
 It's no exaggeration to say the undecideds could go one way or another.  
*George Bush*

Our presidents ... pithy remarks.

```
\begin{quote}
 The buck stops here. \emph{Harry Truman}
 I am not a crook. \emph{Richard Nixon}
 It's no exaggeration ... \emph{George Bush}
\end{quote}
```

The `quotation` environment is used for quotations of more than one paragraph; as usual, the paragraphs are separated by blank lines.

Here is some advice to remember when you are using **L<sup>A</sup>T<sub>E</sub>X**:

Environments for making quotations can be used for other things as well.  
 Many problems can be solved by novel applications of existing environments.

Here is some advice to remember when you are using **\LaTeX**:

```
\begin{quotation}
 Environments for making quotations
 ... other things as well.
 Many ... existing environments.
\end{quotation}
```

## Lists

**L<sup>A</sup>T<sub>E</sub>X** provides three list-making environments: `itemize`, `enumerate`, and `description`. In all three, each new list item is begun with an `\item` command. Itemized lists are made with the `itemize` environment and enumerated lists with the `enumerate` environment.

- Each list item is marked with a *label*. The labels in this itemized list are bullets.
- Lists can be nested within one another.
  1. The item labels in an enumerated list are numerals or letters.
  2. A list should have at least two items.

*L<sup>A</sup>T<sub>E</sub>X* permits at least four levels of nested lists, which is more than enough.

- Blank lines before an item have no effect.

In the `description` environment, you specify the item labels with an optional argument to the `\item` command, enclosed in brackets. (Although the argument is optional, the item will look funny if you omit it.)

Three animals you should know about are:

**gnat** A small animal, found in the North Woods, that causes no end of trouble.

**gnu** A large animal, found in crossword puzzles, that causes no end of trouble.

**armadillo** A medium-sized animal, named after a medium-sized Texas city.

The characters [ and ] are used both to delimit an optional argument and to produce square brackets in the output. This can cause some confusion if the text of an item begins with a [ or if an `\item` command's optional argument contains a square bracket. Section C.1.1 explains what to do in these uncommon situations. All commands that have an optional argument are fragile.

## Poetry

Poetry is displayed with the `verse` environment. A new stanza is begun with one or more blank lines; lines within a stanza are separated by a \\ command.

There is an environment for verse  
 Whose features some poets will curse.  
 For instead of making  
 Them do *all* line breaking,  
 It allows them to put too many words  
 on a line when they'd rather be  
 forced to be terse.

```
\begin{itemize}
\item Each list item is ... bullets.
\item Lists can be ... one another.
\begin{enumerate}
\item The item labels ... letters.
\item A list should ... two items.
\end{enumerate}
\LaTeX\ permits ... more than enough.
\item Blank lines ... have no effect.
\end{itemize}
```

Three animals you should know about are:

```
\begin{description}
\item[gnat] A small animal ...
\item[gnu] A large animal
\item[armadillo] A medium-sized ...
\end{description}
```

```
\begin{verse}
There is an environment for verse \\
Whose features some poets will curse.

For instead of making \\
Them do \emph{all} line breaking, \\
It allows them ... to be terse.
\end{verse}
```

The \\\* command is the same as \\ except that it prevents *L<sup>A</sup>T<sub>E</sub>X* from starting a new page at that point. It can be used to prevent a poem from being

broken across pages in a distracting way. The commands `\\\` and `\\\*` are used in all environments in which you tell L<sup>A</sup>T<sub>E</sub>X where to break lines; several such environments are described in the next chapter. The `\\\*` command is called the *\*-form* of the `\\\` command. Several other commands also have *\*-forms*—versions of the command that are slightly different from the ordinary one—that are obtained by typing `*` after the command name.

The `\\\` and `\\\*` commands have a little-used optional argument described in Section C.1.6, so putting a `[` after them presents the same problem as for the `\item` command. Moreover, the `*` in the `\\\*` command is somewhat like an optional argument for the `\\\` command, so following a `\\\` with a `*` in the text poses a similar problem. See Section C.1.1 for the solutions to these unlikely problems. Almost every command that has a *\*-form* is fragile, and its *\*-form* is also fragile.

### Displayed Formulas

A mathematical formula is displayed when either it is too long to fit comfortably in the running text, it is so important that you want it to stand out, or it is to be numbered for future reference. L<sup>A</sup>T<sub>E</sub>X provides the `displaymath` and `equation` environments for displaying formulas; they are the same except that `equation` numbers the formula and `displaymath` doesn't. Because displayed equations are used so frequently in mathematics, L<sup>A</sup>T<sub>E</sub>X allows you to type `\[ ... \]` instead of

```
\begin{displaymath} ... \end{displaymath}
```

Here is an example of an unnumbered displayed equation:

$$x' + y^2 = z_i^2$$

and here is the same equation numbered:

$$x' + y^2 = z_i^2 \quad (8)$$

Here is an example of an unnumbered displayed equation:

$$\[ x' + y^2 = z_i^2 \]$$

and here is the same equation numbered:

```
\begin{equation}
x' + y^2 = z_i^2
\end{equation}
```

The document class determines how equations are numbered. Section 4.2 describes how L<sup>A</sup>T<sub>E</sub>X can automatically handle references to equation numbers so you don't have to keep track of the numbers.

A displayed formula, like any displayed text, should not begin a paragraph. Moreover, it should not form a complete paragraph by itself. These two observations are summed up in a simple rule: in the input, never leave a blank line before a displayed formula.

T<sub>E</sub>X will not break the formula in a `displaymath` or `equation` environment across lines. See Section 3.3.5 for commands to create a single multiple-line formula or a sequence of displayed formulas.

### 2.2.5 Declarations

You may want to emphasize a large piece of text, such as a quotation. You can do this with the `\emph` command, but that makes the input file hard to read because you have to search for the closing right brace to see where the argument ends. Moreover, it's easy accidentally to delete the closing brace when you edit the text. Instead, you can use the `\em` command, which tells `TeX` to start emphasizing text.

This prose is very dull.

Wait! *Here is an exciting quote.*

Aren't you glad all that excitement is over?

This prose is very dull.

`\begin{quote}`

Wait! `\em` Here is an exciting quote.

`\end{quote}`

Aren't you glad ...

As explained below, the `\end{quote}` caused `TeX` to stop emphasizing text.

Unlike other commands you've encountered so far, `\em` produces neither text nor space; instead, it affects the way `LATEX` prints the text that follows it. Such a command is called a *declaration*. Most aspects of the way `LATEX` formats a document—the type style, how wide the margins are, and so on—are determined by declarations. The `\em` declaration instructs `LATEX` to change the type style to the appropriate one for indicating emphasis. The *scope* of a declaration is ended by an `\end` command or a right brace. In the input, braces and `\begin` and `\end` commands must come in matched pairs. The scope of a declaration is ended by the first `\end` or `}` whose matching `\begin` or `{` precedes the declaration. The following example shows only the braces, `\begin` and `\end` commands (without their arguments and argument-enclosing braces), and an `\em` declaration from some input text; matching braces and matching `\begin` and `\end` commands have the same numbers. The shaded region is the scope of the `\em` declaration.

```
\begin{1} {2} }2 {3} \em [shaded] {4} }4 \end{1}
```

The braces can be the ones that surround a command argument or ones that are inserted just to delimit the scope of a declaration, as in the first emphasized *that* of<sup>5</sup>

That *that that that* thatcher thought thrilling ...      That `\em` *that* that `\emph{that}` ...

The `\{` and `\}` commands do not count for purposes of brace matching and delimiting the scope of declarations. Also, as explained in Section 3.4, argument braces for commands you define yourself do not act as scope delimiters.

Every declaration has a corresponding environment of the same name (minus the `\` character). Typing

<sup>5</sup>You should use the `\emph` command rather than the `\em` declaration for emphasizing small pieces of text because it produces better spacing.

```
\begin{em} ... \end{em}
```

is equivalent to typing `{\em ... }`. (If a declaration has arguments, they become additional arguments of the corresponding environment's `\begin` command.) Using the environment form of a declaration instead of delimiting its scope with braces can make your input file easier to read.

## 2.3 Running L<sup>A</sup>T<sub>E</sub>X

If you followed the directions in Section 1.7, you now know how to run L<sup>A</sup>T<sub>E</sub>X on an input file. If not, consult the *Local Guide* to find out. When you use your own input file for the first time, things are unlikely to go as smoothly as they did for `sample2e.tex`. There will probably be a number of errors in your file—most of them simple typing mistakes. Chapter 8 gives detailed help in diagnosing errors. Here I will tell you how to apply first aid from the keyboard while L<sup>A</sup>T<sub>E</sub>X is still running.

When you start L<sup>A</sup>T<sub>E</sub>X, it will probably be running in one of several windows on your computer's screen. For historical reasons, I will call that window the *terminal*. (If L<sup>A</sup>T<sub>E</sub>X runs by itself and takes up the entire screen, then your whole computer is the terminal.) What you type directly to L<sup>A</sup>T<sub>E</sub>X, and what it types back at you, are called terminal input and output.

With your text editor, produce a new file named `errsam.tex` by making the following four changes to `sample2e.tex`:

- Line 174 of the file contains an `\end{enumerate}` command. Delete the `t` to produce `\end{enumerae}`, simulating a typical typing error.
- Four lines below is an `\item` command. Turn it into `\itemt`, another typical typo.
- The sixth line down from there begins with the word `Whose`. Add a pair of brackets to change that word to `[W]hose`.
- About a dozen lines further down in the file is a line consisting of the single word `is`. Add a space followed by `gnomonly` to the end of that line.

Now run L<sup>A</sup>T<sub>E</sub>X with `errsam.tex` as input and see what error messages it produces. You needn't write them down because everything T<sub>E</sub>X writes on your screen is also written in a file called the *log* file.<sup>6</sup> For the input file `errsam.tex`, the log file is named `errsam.log` on most computers, but it may have a different extension on yours; check your *Local Guide*.

L<sup>A</sup>T<sub>E</sub>X begins by typing pretty much what it did when you ran it on the `sample2e.tex` file, but then writes the following message on your screen and stops.

---

<sup>6</sup>The log file also has some things that don't appear on your screen, including blank lines inserted in strange places.

```
! LaTeX Error: \begin{enumerate} on input line 167 ended by \end{enumerae}.
```

```
See the LaTeX manual or LaTeX Companion for explanation.
```

```
Type H <return> for immediate help.
```

```
...
```

```
1.174 \end{enumerae}
```

```
?
```

L<sup>A</sup>T<sub>E</sub>X translates a command like \end, which describes the document's logical structure, into T<sub>E</sub>X's typesetting commands. Some errors are caught by L<sup>A</sup>T<sub>E</sub>X; others cause it to generate typesetting commands containing errors that T<sub>E</sub>X finds. The first line of this message, called the *error indicator*, tells us that the error was found by L<sup>A</sup>T<sub>E</sub>X rather than T<sub>E</sub>X. Like all error messages, this one begins with an exclamation point.

The error indicator tells what the problem is. Chapter 8 explains the meaning of the error indicators for most L<sup>A</sup>T<sub>E</sub>X-detected errors and for the most common errors that T<sub>E</sub>X finds. Here, L<sup>A</sup>T<sub>E</sub>X is complaining that the \end command that matches the \begin{enumerate} is not an \end{enumerae}, as it should be. The line beginning with 1.174 is the *error locator*, telling you where in your input file the error was discovered. In this case, it was on line 174, after T<sub>E</sub>X read the \end{enumerae} command. The ? that ends the message indicates that L<sup>A</sup>T<sub>E</sub>X has stopped and is waiting for you to type something.

Just pass over the error by pressing the *return* key, which instructs L<sup>A</sup>T<sub>E</sub>X to continue processing the input. L<sup>A</sup>T<sub>E</sub>X now writes

```
! Undefined control sequence.
```

```
1.178 \itemt
```

```
 This is the third item of the list.
```

```
?
```

The absence of the L<sup>A</sup>T<sub>E</sub>X Error at the beginning of the message tells you that this error was detected by T<sub>E</sub>X rather than L<sup>A</sup>T<sub>E</sub>X. T<sub>E</sub>X knows nothing about L<sup>A</sup>T<sub>E</sub>X commands, so you can't expect much help from the error indicator. The error locator indicates that the error was detected on line 178 of the input file, after reading \itemt and before reading This. (The error indicator line is broken at the point where T<sub>E</sub>X stopped reading input.) Of course, this error is caused by \itemt, which is a command name T<sub>E</sub>X has never heard of.

Continue past this error by pressing *return*. L<sup>A</sup>T<sub>E</sub>X next writes

```
! Missing number, treated as zero.
<to be read again>
```

```
W
```

```
1.184 [W]
```

```
 hose features some poets % within a stanza.
```

```
?
```



This error was found by  $\text{\TeX}$  not  $\text{\LaTeX}$ , and the error indicator isn't very helpful.  $\text{\TeX}$  has just read the [W], which looks all right, so the error must have been caused by something that came before it. Right before the [W] in the input file is a  $\backslash\backslash$  command. Do you now see what the error is? If not, look in the index under " $\backslash\backslash$ , [ after".

Press *return* to continue. You immediately get another error message very similar to the preceding one, with the same error locator.  $\text{\TeX}$  has discovered a second error in the low-level typesetting commands generated by the  $\backslash\backslash$  command. Type *return* again to get past this problem.

When you find the first error, it's tempting to stop the program, correct it, and start over again. However, if you've made one error, you've probably made more. It's a good idea to find several errors at once, rather than running  $\text{\LaTeX}$  over and over to find one mistake at a time. Keep typing *return* and try to get as far as you can. You may reach an impasse. A single mistake can cause  $\text{\TeX}$  to produce hundreds of error messages, or to keep generating the same message over and over again forever. You can stop  $\text{\TeX}$  before it's finished by typing X followed by *return* in response to an error message.

$\text{\TeX}$  may write a \* and stop without any error message. This is probably due to a missing `\end{document}` command, but other errors can also cause it. If it happens, type `\stop` followed by *return*. If that doesn't work, you'll have to use your computer's standard method for halting recalcitrant programs, which is described in the *Local Guide*.

Instead of sitting at your keyboard waiting for errors, you can let  $\text{\LaTeX}$  run unattended and find out what happened later by reading the log file. A `\batchmode` command at the very beginning of the input file causes  $\text{\TeX}$  to process the file without stopping—much as if you were to type *return* in response to every error message, except the messages are not actually written on your screen. This is a convenient way to run  $\text{\LaTeX}$  while you go out to lunch, but you could return to find that a small error resulted in a very long list of error messages in the log file.

Meanwhile,  $\text{\LaTeX}$  has finished processing your input file `errsam.tex`. After the last error message, it wrote

```
Overfull \hbox (10.51814pt too wide) in paragraph at lines 195--200
[]\OT1/cmr/m/n/10 Mathematical for-mu-las may also be dis-played. A dis-played
for-mula is gnomonly
```

This is a *warning* message;  $\text{\LaTeX}$  did not stop (it didn't print a ?), but continued to the end of the input file without further incident.  $\text{\TeX}$  generated the warning because it does not know how to hyphenate the word *gnomonly*, so it couldn't find a good place to break the line. If you look at the output, you'll find *gnomonly* extending beyond the right margin. This is not a serious problem; Section 6.2.1 describes how to correct it.

When you process your input file for the first time,  $\text{\TeX}$  is likely to produce lots of error messages and warnings that you may not understand right away.

The most important thing to remember is, DON'T PANIC. Instead, turn to Chapter 8 to find out what to do.

## 2.4 Helpful Hints

The descriptions of individual  $\text{\LaTeX}$  features include suggestions about their use. Here are a few general recommendations that can make your life easier.

As soon as you have some experience with  $\text{\LaTeX}$ , read Section 3.4 to learn how to define your own commands and environments. When I write a paper, I change my notation much more than I change my concepts. Defining commands to express concepts allows me to change notation by simply modifying the command definitions, without having to change every occurrence in the text. This saves a lot of work.

To avoid errors and simplify making changes, keep your input file as easy to read as possible. Spacing and indentation can help. Use comments, especially to explain your command definitions.  $\text{\TeX}$  doesn't care how the input file is formatted, but you should.

As you write your document, you will be continually running  $\text{\LaTeX}$  on it so you can view the latest version. As the document gets longer,  $\text{\LaTeX}$  takes longer to process it. When you make some changes to Section 7, don't run  $\text{\LaTeX}$  on Sections 1–6 just to find errors in the new material. Find those errors fast by running  $\text{\LaTeX}$  only on Section 7. Process the entire document only when you're pretty sure Section 7 has no more errors.

There are two ways to run  $\text{\LaTeX}$  only on what's changed. The first is to create a new file containing just the preamble, a `\begin{document}` command, the new material, and an `\end{document}` command. This is easy to do with a good text editor. I have programmed my editor so, at the stroke of a couple of keys, it will create this file, run  $\text{\LaTeX}$  on it, and display the output with a screen previewer—while I keep editing my document. The second way to run  $\text{\LaTeX}$  just on what has changed is to keep your input in several files, using the commands of Section 4.4 to process one part at a time. Experiment to find out what method works best for you.

Perhaps the most annoying aspect of a computer program is the way it reacts to your errors. As with most programs,  $\text{\LaTeX}$ 's train of thought is derailed by simple errors that any person would easily correct. The best way to avoid this problem is to avoid those simple errors. Here are some common ones that are easy to eliminate by being careful:

- A misspelled command or environment name.
- Improperly matching braces.
- Trying to use one of the ten special characters `# $ % & _ { } ^ ~ \` as an ordinary printing symbol.

- Improperly matching formula delimiters—for example, a `\(`` command without the matching `\)``.
- The use in ordinary text of a command like `^` that can appear only in a mathematical formula.
- A missing `\end` command.
- A missing command argument.

A good text editor can detect or help prevent some of these errors.

Because of a bit of fossilized stupidity, e-mail programs on the Unix operating system often add a `>` character to the beginning of every line that begins with the five characters `From`<sub>U</sub>. The input `>From` causes TeX to produce the output `„From`. It's a good idea to search all L<sup>A</sup>T<sub>E</sub>X files received by e-mail for `>From`, and to add a space to the beginning of every line that starts with `From` in all files that you send by e-mail.

## 2.5 Summary

This chapter has explained everything you have to know to prepare a simple document. There is quite a bit to remember. Here is a summary to refresh your memory.

### Input Characters

The input file may contain the following characters: uppercase and lowercase letters, the ten digits, the sixteen punctuation characters

`. : ; , ? ! ' ' ( ) [ ] - / * @`

the ten special characters

`# $ % & _ { } ^ ^ \`

(the first seven are printed by the commands `\#`, `\$`, etc.), and the five characters `+ = | < >` used mainly in mathematical formulas. There are also invisible characters, which are all denoted by `U`, that produce spaces in the input file.

### Commands and Environments

Command names consist of either a single special character like `~`, a `\` followed by a single nonletter (as in `\@`), or a `\` followed by a string of letters. Spaces and a single end-of-line following the latter kind of command name are ignored; use a `\_` command to put an interword space after such a command. The case of letters in command names counts; most L<sup>A</sup>T<sub>E</sub>X command names contain only

lowercase letters. A few commands have a *\*-form*, a variant obtained by typing *\** after the command name.

Command arguments are enclosed in curly braces `{` and `}`, except for optional arguments, which are enclosed in square brackets `[` and `]`. See Section C.1.1 if an optional argument has a square bracket or if a `[` in the text could be confused with the start of an optional argument. Do not leave any extra space within an argument; use a `\%` to end a line without introducing space.

Some commands have *moving arguments*. The name of a fragile command must be preceded by a `\protect` command when it appears in a moving argument. Fragile commands include `\(`, `\)`, `\[`, `\]`, `\begin`, `\end`, `\backslash`, `\item`, and `\footnote`. A `\protect` command seldom hurts; when in doubt use one.

A declaration is a command that directs  $\text{\LaTeX}$  to change the way it is formatting the document. The scope of a declaration is delimited by enclosing it within curly braces or within an environment.

An environment has the form

```
\begin{name} ... \end{name}
```

To every declaration corresponds an environment whose name is the same as the declaration's name without the `\`.

## Sentences and Paragraphs

Sentences and paragraphs are typed pretty much as expected.  $\text{\TeX}$  ignores the formatting of the input file. A blank line indicates a new paragraph.

Quotes are typed with the `'` and `'` characters, used in pairs for double quotes. The `\` command separates multiple quotation marks, as in `“\,‘Fum’\,‘”`.

Dashes of various sizes are produced with one, two, or three `“-”` characters.

A period, question mark, or exclamation point is considered to end a sentence unless it follows an uppercase letter. A `\@` command before the punctuation character forces  $\text{\TeX}$  to treat it as the end of a sentence, while a `\_` command placed after it produces an interword space.

The  $\text{\TeX}$  and  $\text{\LaTeX}$  logos are produced by the `\TeX` and `\LaTeX` commands. The `\today` command produces the current date, and `\ldots` produces an ellipsis `(...)`.

Text is emphasized with the `\emph` command.

The `\~` command produces an interword space at which  $\text{\TeX}$  will not start a new line. The `\mbox` command prevents  $\text{\TeX}$  from breaking its argument across lines.

Footnotes are typed with the `\footnote` command, whose argument is the text of the footnote.

In-line mathematical formulas are enclosed by `\( ... \)` or `$ ... $`. Subscripts and superscripts are made with the `_` and `^` commands. The `'` character produces a prime symbol `(')`.

## Larger Structures

The document begins with the preamble. The preamble begins (after any prepended files—see Section 4.7) with a `\documentclass` command, which may be followed by one or more `\usepackage` commands. The rest of the preamble may contain command definitions and special style declarations for the document. The actual text is contained in a `document` environment.

A title is produced by using the `\title`, `\author`, and `\date` commands to declare the necessary information, and the `\maketitle` command to generate the title. Multiple authors are separated by `\and` commands in the argument of `\author`.

A sectional unit is begun with one of the following sectioning commands

|                       |                             |                            |
|-----------------------|-----------------------------|----------------------------|
| <code>\part</code>    | <code>\subsection</code>    | <code>\paragraph</code>    |
| <code>\chapter</code> | <code>\subsubsection</code> | <code>\subparagraph</code> |
| <code>\section</code> |                             |                            |

whose argument produces the unit's heading and is a moving argument.

## Displayed Material

Short quotations are displayed with the `quote` environment and long quotations with the `quotation` environment.

`LATEX` provides three list-making environments: `itemize` for itemized lists, `enumerate` for enumerated lists, and `description` for lists with user-specified labels. Each item is begun with an `\item` command whose optional argument provides the item labels in the `description` environment.

The `verse` environment is used for poetry. A blank line begins a new stanza, and a line that does not end a stanza is followed by a `\\\` command—use `\\\*` instead of `\\\` to prevent a page break after the line. (See Section C.1.1 if a `*` follows an ordinary `\\\` command.)

Displayed mathematical formulas are produced with the `displaymath` environment or the equivalent `\[ ... \]` construction. The `equation` environment produces numbered displayed formulas.

## CHAPTER 3

# Carrying On



Chapter 2 described commands for simple documents. Sooner or later, you'll write something that requires more sophisticated formatting. The commands and environments described in this chapter will handle most of those situations. Before getting to them, you should know a little more about how  $\text{\TeX}$  operates.

As  $\text{\TeX}$  processes your input text, it is always in one of three *modes*: paragraph mode, math mode, or left-to-right mode (called LR mode for short).<sup>1</sup> Paragraph mode is  $\text{\TeX}$ 's normal mode—the one it's in when processing ordinary text. In paragraph mode,  $\text{\TeX}$  regards your input as a sequence of words and sentences to be broken into lines, paragraphs, and pages.

$\text{\TeX}$  is in math mode when it's generating a mathematical formula. More precisely, it enters math mode upon encountering a command like  $\$$  or  $\backslash ($  or  $\backslash [$  or  $\backslash \begin{equation}$  that begins a mathematical formula, and it leaves math mode after finding the corresponding command like  $\backslash )$  that ends the formula. When  $\text{\TeX}$  is in math mode, it considers letters in the input file to be mathematical symbols, treating  $is$  as the product of  $i$  and  $s$ , and ignores any space characters between them.

In LR mode, as in paragraph mode,  $\text{\TeX}$  considers your input to be a string of words with spaces between them. However, unlike paragraph mode,  $\text{\TeX}$  produces output that keeps going from left to right; it never starts a new line in LR mode. The  $\backslash \mbox$  command (Section 2.2.1) causes  $\text{\TeX}$  to process its argument in LR mode, which is what prevents the argument from being broken across lines.

Different modes can be nested within one another. If you put an  $\backslash \mbox$  command inside a mathematical formula,  $\text{\TeX}$  is in LR mode when processing that command's argument, not in math mode. In the example

$y > z$  if  $x^2$  real.

$\backslash ( y > z \backslash \mbox{ if } \$x^{\{2\}}\$ \text{ real} ) \backslash )$ .

$\text{\TeX}$  is in math mode when processing  $y$  and  $z$ , in LR mode when processing  $\text{if}$  and  $\text{real}$ , and in math mode when processing  $x^2$ . The space between “ $z$ ” and “if” is produced by the first  $\backslash$  in the  $\backslash \mbox$  command's argument, since space characters in the input produce space in the output when  $\text{\TeX}$  is in LR mode. The  $\backslash$  in  $\text{real}\backslash )$  is processed in math mode, so it produces no space between “real” and “.” in the output.

### 3.1 Changing the Type Style

Type style is used to indicate logical structure. In this book, emphasized text appears in *italic* style type and  $\text{\LaTeX}$  input in *typewriter* style. In  $\text{\LaTeX}$ , a type style is specified by three components: shape, series, and family.

---

<sup>1</sup>Paragraph mode corresponds to the vertical and ordinary horizontal modes in *The TeXbook*, and LR mode is called restricted horizontal mode there.  $\text{\LaTeX}$  also has a restricted form of LR mode called *picture* mode that is described in Section 7.1.

Upright shape. Usually the default.

```
\textup{Upright shape...}
```

*Italic shape.* Often used for emphasized text.

```
\textit{Italic shape...}
```

Slanted shape. A bit different from italic.

```
\textsl{Slanted shape...}
```

SMALL CAPS SHAPE. USE SPARINGLY.

```
\textsc{Small caps shape...}
```

Medium series. Usually the default.

```
\textmd{Medium series...}
```

**Boldface series.** Often used for headings.

```
\textbf{Boldface series...}
```

Roman family. Usually the default.

```
\textrm{Roman family...}
```

Sans serif family. Often used in ads.

```
\textsf{Sans serif family...}
```

Typewriter family. Popular with hackers.

```
\texttt{Typewriter family...}
```

These commands can be combined in a logical fashion to produce a wide variety of type styles.

**Who on Earth** is *ever* going to use a boldface sans serif or an italic typewriter type style?

```
\textsf{\textbf{Who \textmd{on} Earth}} is
\textit{\texttt{ever}} ...
```

Some type styles may be unavailable on your computer. If you specify a style that isn't available, L<sup>A</sup>T<sub>E</sub>X will write a warning message and substitute a style that it thinks is similar. (You may think otherwise.)

Each of the text-style commands described above has a corresponding declaration. Boldface text can be obtained with either the `\textbf` text-producing command or the `\bfseries` declaration.

**More** and **more** armadillos are crossing the road. `\textbf{More}` and `\bfseries more` ...

The declarations corresponding to the text-producing command are:

| <i>cmd</i>           | <i>decl</i>           | <i>cmd</i>           | <i>decl</i>            | <i>cmd</i>           | <i>decl</i>            |
|----------------------|-----------------------|----------------------|------------------------|----------------------|------------------------|
| <code>\textup</code> | <code>\upshape</code> | <code>\textsc</code> | <code>\scshape</code>  | <code>\textrm</code> | <code>\rmfamily</code> |
| <code>\textit</code> | <code>\itshape</code> | <code>\textmd</code> | <code>\mdseries</code> | <code>\textsf</code> | <code>\sffamily</code> |
| <code>\textsl</code> | <code>\slshape</code> | <code>\textbf</code> | <code>\bfseries</code> | <code>\texttt</code> | <code>\ttfamily</code> |

None of these text-producing commands or declarations can be used in math mode. Section 3.3.8 explains how to change type style in a mathematical formula.

Type style is a visual property. Commands to specify visual properties belong not in the text, but in the definitions of commands that describe logical structure. L<sup>A</sup>T<sub>E</sub>X provides the `\emph` command for emphasized text; Section 3.4 explains how to define your own commands for the logical structures in your document. For example, suppose you want the names of genera to appear in italic in your book on African mammals. Don't use `\textit` throughout the text; instead, define a `\genus` command and write

*Connochaetes* seems to pop up everywhere.

```
\genus{Connochaetes} seems to pop up ...
```

Then, if you decide that *Connochaetes* and all other genera should appear in slanted rather than italic type, you just have to change the definition of `\genus`.

## 3.2 Symbols from Other Languages

Languages other than English have a variety of accents and special symbols. This section tells you how to generate the ones used in most Western languages. These accents and symbols are not available in the typewriter family of type styles. All the commands introduced in this section are robust.

Commands to produce accents and symbols from other languages allow you to put small pieces of non-English text in an English document. They are not adequate for writing a complete document in another language. The `babel` package allows you to produce documents in languages other than English, as well as multilanguage documents. Consult the *L<sup>A</sup>T<sub>E</sub>X Companion* for details.

### 3.2.1 Accents

Table 3.1 shows how to make a wide variety of accents. In this and all similar tables, the `TeX` output is followed by the input that produces it, the first entry in Table 3.1 showing that you produce  $\circ$  by typing `\{'o}`. The letter `o` appears in this table, but the commands can accent any letter.

El señor está bien, garçon.

El se\~{n}or est\`{a} bien, gar\c{c}on.

The letters *i* and *j* need special treatment because they should lose their dots when accented. The commands `\i` and `\j` produce a dotless *i* and *j*, respectively.

Él está aquí.

\'{E}l est\`{a} aqu\`{i}.

The commands in Table 3.1 can be used only in paragraph and LR modes. Accents in math mode, which produce accented symbols in mathematical formulas, are made with commands described in Section 3.3.6.

### 3.2.2 Symbols

Table 3.2 shows how to make some symbols from non-English languages. Note that the symbols ; and ; are produced by typing a pair of punctuation characters, in much the same way that a medium-length dash is produced by typing two – characters. The commands in Table 3.2 can appear only in paragraph and LR modes; use an `\mbox` command to put one inside a mathematical formula.

|                   |                   |                   |                  |
|-------------------|-------------------|-------------------|------------------|
| $\circ$ \{'o}     | $\tilde{o}$ \~{o} | $\ddot{o}$ \v{o}  | $\ddot{o}$ \c{o} |
| $\bar{o}$ \'{o}   | $\bar{o}$ \={o}   | $\ddot{o}$ \H{o}  | $\bar{o}$ \d{o}  |
| $\hat{o}$ \^{o}   | $\hat{o}$ \.{o}   | $\ddot{o}$ \t{oo} | $\bar{o}$ \b{o}  |
| $\ddot{o}$ \\"{o} | $\ddot{o}$ \u{o}  |                   |                  |

Table 3.1: Accents.

|                 |                 |                 |              |
|-----------------|-----------------|-----------------|--------------|
| $\oe$ \oe       | $\aa$ \aa       | $\l$ \l         | $\text{?}$ ? |
| $\text{OE}$ \OE | $\text{AA}$ \AA | $\text{L}$ \L   | $\text{!}$ ! |
| $\ae$ \ae       | $\text{o}$ \o   | $\text{ss}$ \ss |              |
| $\text{AE}$ \AE | $\text{O}$ \O   |                 |              |

Table 3.2: Non-English Symbols.

The following six special punctuation symbols can be used in any mode:

|               |                       |                                     |
|---------------|-----------------------|-------------------------------------|
| $\dag$ \dag   | $\S$ \S               | $\text{\textcircled{C}}$ \copyright |
| $\ddag$ \ddag | $\text{\textperp}$ \P | $\text{\textpounds}$ \pounds        |

Remember also that the seven symbols  $\#$   $\%$   $\&$   $_$   $\{$   $\}$  are produced by the seven commands  $\#$   $\$$   $\%$   $\&$   $_$   $\{$   $\}$ .

In addition to the symbol-making commands described here, there are many others that can be used only in math mode. They are described in Section 3.3.2.

### 3.3 Mathematical Formulas

A formula that appears in the running text, called an *in-text* formula, is produced by the **math** environment. This environment can be invoked with either of the two short forms  $\(...\)$  or  $$...$$ , as well as by the usual  $\begin{...}$   $\end{...}$  construction. The **displaymath** environment, which has the short form  $\[...]$ , produces an unnumbered displayed formula. The short forms  $$...$$ ,  $\(...\)$ , and  $\[...]$  act as full-fledged environments, delimiting the scope of declarations contained within them. A numbered displayed formula is produced by the **equation** environment. Section 4.2 describes commands for assigning names to equation numbers and referring to the numbers by name, so you don't have to keep track of the actual numbers.

The **math**, **displaymath**, and **equation** environments put **TeX** in math mode. **TeX** ignores spaces in the input when it's in math mode (but space characters may still be needed to mark the end of a command name). Section 3.3.7 describes how to add and remove space in formulas. Remember that **TeX** is in LR mode, where spaces in the input generate space in the output, when it begins processing the argument of an **\mbox** command—even one that appears inside a formula.

All the commands introduced in this section can be used only in math mode, unless it is explicitly stated that they can be used elsewhere. Except as noted, they are all robust. However, **\begin**, **\end**, **\(**, **\)**, **\[**, and **\]** are fragile commands.

### 3.3.1 Some Common Structures

#### Subscripts and Superscripts

Subscripts and superscripts are made with the `_` and `^` commands. These commands can be combined to make complicated subscript and superscript expressions.

|          |                     |           |                      |             |                          |
|----------|---------------------|-----------|----------------------|-------------|--------------------------|
| $x^{2y}$ | <code>x^{2y}</code> | $x^{y^2}$ | <code>x^{y^2}</code> | $x_1^y$     | <code>x^y_{1}</code>     |
| $x_{2y}$ | <code>x_{2y}</code> | $x^{y_1}$ | <code>x^{y_1}</code> | $x_1^{y_1}$ | <code>x^{y_1}_{1}</code> |

#### Fractions

Fractions denoted by the `/` symbol are made in the obvious way.

Multiplying by  $n/2$  gives  $(m+n)/n$ .

Multiplying by `$n/2$` gives `\(( (m+n)/n )\)`.

Most fractions in the running text are written this way. The `\frac` command is used for large fractions in displayed formulas; it has two arguments: the numerator and denominator.

$$x = \frac{y + z/2}{y^2 + 1}$$

$$\frac{x + y}{1 + \frac{y}{z+1}}$$

`\[ x = \frac{y+z/2}{y^2+1} \]`

`\[\frac{x+y}{1+\frac{y}{z+1}}\]`

The `\frac` command can be used in an in-text formula to produce a fraction like  $\frac{1}{2}$  (by typing `\frac{1}{2}`), but this is seldom done.

#### Roots

The `\sqrt` command produces the square root of its argument; it has an optional first argument for other roots. It is a fragile command.

A square root  $\sqrt{x+y}$  and an  $n$ th root  $\sqrt[n]{2}$ .

`... \(\sqrt{x+y}\) ... \(\sqrt[n]{2}\)`.

#### Ellipsis

The commands `\ldots` and `\cdots` produce two different kinds of ellipsis (...).

A low ellipsis:  $x_1, \dots, x_n$ .

A low ellipsis: `$x_{1}, \ldots, x_{n}$`.

A centered ellipsis:  $a + \cdots + z$ .

A centered ellipsis: `$a + \cdots + z$`.

Use `\ldots` between commas and between juxtaposed symbols like  $a \dots z$ ; use `\cdots` between symbols like  $+$ ,  $-$ , and  $=$ . `\TeX` can also produce vertical and diagonal ellipses, which are used mainly in arrays.

⋮ \vdots ⋯ \cdots ⋮ \vdots ⋯ \ddots

The `\ldots` command works in any mode, but `\cdots`, `\vdots`, and `\ddots` can be used only in math mode.

### 3.3.2 Mathematical Symbols

There are TeX commands to make almost any mathematical symbol you're likely to need. Remember that they can be used only in math mode.

#### Greek Letters

The command to produce a lowercase Greek letter is obtained by adding a `\` to the name of the letter. For an uppercase Greek letter, just capitalize the first letter of the command name.

Making Greek letters is as easy as  $\pi$  (or  $\Pi$ ). ... is as easy as `\pi` (or `\Pi`).

(The `$`'s are needed because these commands can be used only in math mode.) If the uppercase Greek letter is the same as its Roman equivalent, as in uppercase alpha, then there is no command to generate it. A complete list of the commands for making Greek letters appears in Table 3.3. Note that some of the lowercase letters have variant forms, made by commands beginning with `\var`. Also, observe that there's no special command for an omicron; you just use an `o`.

| Lowercase                 |                       |                       |                     |
|---------------------------|-----------------------|-----------------------|---------------------|
| $\alpha$ \alpha           | $\theta$ \theta       | $\circ$ o             | $\tau$ \tau         |
| $\beta$ \beta             | $\vartheta$ \vartheta | $\pi$ \pi             | $\upsilon$ \upsilon |
| $\gamma$ \gamma           | $\iota$ \iota         | $\varpi$ \varpi       | $\phi$ \phi         |
| $\delta$ \delta           | $\kappa$ \kappa       | $\rho$ \rho           | $\varphi$ \varphi   |
| $\epsilon$ \epsilon       | $\lambda$ \lambda     | $\varrho$ \varrho     | $\chi$ \chi         |
| $\varepsilon$ \varepsilon | $\mu$ \mu             | $\sigma$ \sigma       | $\psi$ \psi         |
| $\zeta$ \zeta             | $\nu$ \nu             | $\varsigma$ \varsigma | $\omega$ \omega     |
| $\eta$ \eta               | $\xi$ \xi             |                       |                     |
| Uppercase                 |                       |                       |                     |
| $\Gamma$ \Gamma           | $\Lambda$ \Lambda     | $\Sigma$ \Sigma       | $\Psi$ \Psi         |
| $\Delta$ \Delta           | $\Xi$ \Xi             | $\Upsilon$ \Upsilon   | $\Omega$ \Omega     |
| $\Theta$ \Theta           | $\Pi$ \Pi             | $\Phi$ \Phi           |                     |

Table 3.3: Greek Letters.

## Calligraphic Letters

TeX provides twenty-six uppercase calligraphic letters  $\mathcal{A}, \mathcal{B}, \dots, \mathcal{Z}$ , also called script letters. They are produced by a special type style invoked with the `\mathcal` command.

Choose  $\mathcal{F}$  such that  $\mathcal{F}(x) > 0$ .

Choose `\mathcal{F}` such that...

Only the twenty-six uppercase letters are available in the calligraphic type style.

## A Menagerie of Mathematical Symbols

TeX can make dozens of special mathematical symbols. A few of them, such as  $+$  and  $>$ , are produced by typing the corresponding keyboard character. Others are obtained with the commands in Tables 3.4 through 3.7. The shaded symbols require the `latexsym` package to be loaded with a `\usepackage` command. (See Section 2.2.2.) Additional symbols can be made by stacking one symbol on top of another with the `\stackrel` command of Section 3.3.6 or the `array` environment of Section 3.3.3. You can also put a slash through a symbol by typing `\not` before it.

If  $x \not< y$  then  $x \not\leq y - 1$ .

If  $x \not< y$  then  $\not( x \not\leq y - 1 )$ .

If the slash doesn't come out in exactly the right spot, put one of the math-mode spacing commands described in Section 3.3.7 between the `\not` command and the symbol.

There are some mathematical symbols whose size depends upon what kind of math environment they appear in; they are bigger in the `displaymath` and `equation` environments than in the ordinary `math` environment. These symbols are listed in Table 3.8, where both the large and small versions are shown. Subscript-sized expressions that appear above and below them are typed as ordinary subscripts and superscripts.

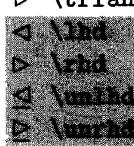
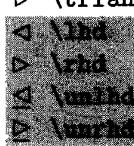
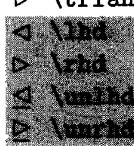
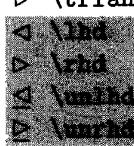
|           |                      |             |                        |                                                                                     |                               |            |                       |
|-----------|----------------------|-------------|------------------------|-------------------------------------------------------------------------------------|-------------------------------|------------|-----------------------|
| $\pm$     | <code>\pm</code>     | $\cap$      | <code>\cap</code>      | $\diamond$                                                                          | <code>\diamond</code>         | $\oplus$   | <code>\oplus</code>   |
| $\mp$     | <code>\mp</code>     | $\cup$      | <code>\cup</code>      | $\triangle$                                                                         | <code>\bigtriangleup</code>   | $\ominus$  | <code>\ominus</code>  |
| $\times$  | <code>\times</code>  | $\uplus$    | <code>\uplus</code>    | $\triangledown$                                                                     | <code>\bigtriangledown</code> | $\otimes$  | <code>\otimes</code>  |
| $\div$    | <code>\div</code>    | $\sqcap$    | <code>\sqcap</code>    | $\triangleleft$                                                                     | <code>\triangleleft</code>    | $\oslash$  | <code>\oslash</code>  |
| $*$       | <code>\ast</code>    | $\sqcup$    | <code>\sqcup</code>    | $\triangleright$                                                                    | <code>\triangleright</code>   | $\odot$    | <code>\odot</code>    |
| $\star$   | <code>\star</code>   | $\vee$      | <code>\vee</code>      |  | <code>\diamond</code>         | $\bigcirc$ | <code>\bigcirc</code> |
| $\circ$   | <code>\circ</code>   | $\wedge$    | <code>\wedge</code>    |  | <code>\triangle</code>        | $\dagger$  | <code>\dagger</code>  |
| $\bullet$ | <code>\bullet</code> | $\setminus$ | <code>\setminus</code> |  | <code>\triangle</code>        | $\ddagger$ | <code>\ddagger</code> |
| $\cdot$   | <code>\cdot</code>   | $\wr$       | <code>\wr</code>       |  | <code>\triangle</code>        | $\amalg$   | <code>\amalg</code>   |

Table 3.4: Binary Operation Symbols. (Shaded ones require `latexsym` package.)

|               |                          |               |                          |           |                      |             |                        |
|---------------|--------------------------|---------------|--------------------------|-----------|----------------------|-------------|------------------------|
| $\leq$        | <code>\leq</code>        | $\geq$        | <code>\geq</code>        | $\equiv$  | <code>\equiv</code>  | $\models$   | <code>\models</code>   |
| $\prec$       | <code>\prec</code>       | $\succ$       | <code>\succ</code>       | $\sim$    | <code>\sim</code>    | $\perp$     | <code>\perp</code>     |
| $\preceq$     | <code>\preceq</code>     | $\succeq$     | <code>\succeq</code>     | $\simeq$  | <code>\simeq</code>  | $\mid$      | <code>\mid</code>      |
| $\ll$         | <code>\ll</code>         | $\gg$         | <code>\gg</code>         | $\asymp$  | <code>\asymp</code>  | $\parallel$ | <code>\parallel</code> |
| $\subset$     | <code>\subset</code>     | $\supset$     | <code>\supset</code>     | $\approx$ | <code>\approx</code> | $\bowtie$   | <code>\bowtie</code>   |
| $\subseteq$   | <code>\subseteq</code>   | $\supseteq$   | <code>\supseteq</code>   | $\cong$   | <code>\cong</code>   | $\Join$     | <code>\Join</code>     |
| $\sqsubset$   | <code>\sqsubset</code>   | $\sqsupset$   | <code>\sqsupset</code>   | $\neq$    | <code>\neq</code>    | $\smile$    | <code>\smile</code>    |
| $\sqsubseteq$ | <code>\sqsubseteq</code> | $\sqsupseteq$ | <code>\sqsupseteq</code> | $\doteq$  | <code>\doteq</code>  | $\frown$    | <code>\frown</code>    |
| $\in$         | <code>\in</code>         | $\ni$         | <code>\ni</code>         | $\notin$  | <code>\notin</code>  | $\propto$   | <code>\propto</code>   |
| $\vdash$      | <code>\vdash</code>      | $\dashv$      | <code>\dashv</code>      |           |                      |             |                        |

Table 3.5: Relation Symbols. (Shaded ones require `latexsym` package.)

|                         |                                    |                             |                                        |                |                           |
|-------------------------|------------------------------------|-----------------------------|----------------------------------------|----------------|---------------------------|
| $\leftarrow$            | <code>\leftarrow</code>            | $\longleftarrow$            | <code>\longleftarrow</code>            | $\uparrow$     | <code>\uparrow</code>     |
| $\Leftarrow$            | <code>\Leftarrow</code>            | $\Longleftarrow$            | <code>\Longleftarrow</code>            | $\Uparrow$     | <code>\Uparrow</code>     |
| $\rightarrow$           | <code>\rightarrow</code>           | $\longrightarrow$           | <code>\longrightarrow</code>           | $\downarrow$   | <code>\downarrow</code>   |
| $\Rightarrow$           | <code>\Rightarrow</code>           | $\Longrightarrow$           | <code>\Longrightarrow</code>           | $\Downarrow$   | <code>\Downarrow</code>   |
| $\leftarrow\rightarrow$ | <code>\leftarrow\rightarrow</code> | $\longleftarrow\rightarrow$ | <code>\longleftarrow\rightarrow</code> | $\updownarrow$ | <code>\updownarrow</code> |
| $\Leftarrow\rightarrow$ | <code>\Leftarrow\rightarrow</code> | $\Longleftarrow\rightarrow$ | <code>\Longleftarrow\rightarrow</code> | $\Updownarrow$ | <code>\Updownarrow</code> |
| $\mapsto$               | <code>\mapsto</code>               | $\longmapsto$               | <code>\longmapsto</code>               | $\nearrow$     | <code>\nearrow</code>     |
| $\hookleftarrow$        | <code>\hookleftarrow</code>        | $\hookrightarrow$           | <code>\hookrightarrow</code>           | $\searrow$     | <code>\searrow</code>     |
| $\leftharpoonup$        | <code>\leftharpoonup</code>        | $\rightharpoonup$           | <code>\rightharpoonup</code>           | $\swarrow$     | <code>\swarrow</code>     |
| $\leftharpoonondown$    | <code>\leftharpoonondown</code>    | $\rightharpoonondown$       | <code>\rightharpoonondown</code>       | $\nwarrow$     | <code>\nwarrow</code>     |
| $\rightleftharpoons$    | <code>\rightleftharpoons</code>    | $\leadsto$                  | <code>\leadsto</code>                  |                |                           |

Table 3.6: Arrow Symbols. (Shaded ones require `latexsym` package.)

|          |                     |             |                        |              |                         |                |                           |
|----------|---------------------|-------------|------------------------|--------------|-------------------------|----------------|---------------------------|
| $\aleph$ | <code>\aleph</code> | $\prime$    | <code>\prime</code>    | $\forall$    | <code>\forall</code>    | $\infty$       | <code>\infty</code>       |
| $\hbar$  | <code>\hbar</code>  | $\emptyset$ | <code>\emptyset</code> | $\exists$    | <code>\exists</code>    | $\Box$         | <code>\Box</code>         |
| $\imath$ | <code>\imath</code> | $\nabla$    | <code>\nabla</code>    | $\neg$       | <code>\neg</code>       | $\Diamond$     | <code>\Diamond</code>     |
| $\jmath$ | <code>\jmath</code> | $\surd$     | <code>\surd</code>     | $\flat$      | <code>\flat</code>      | $\triangle$    | <code>\triangle</code>    |
| $\ell$   | <code>\ell</code>   | $\top$      | <code>\top</code>      | $\natural$   | <code>\natural</code>   | $\clubsuit$    | <code>\clubsuit</code>    |
| $\wp$    | <code>\wp</code>    | $\bot$      | <code>\bot</code>      | $\sharp$     | <code>\sharp</code>     | $\diamondsuit$ | <code>\diamondsuit</code> |
| $\Re$    | <code>\Re</code>    | $\setminus$ | <code>\setminus</code> | $\backslash$ | <code>\backslash</code> | $\heartsuit$   | <code>\heartsuit</code>   |
| $\Im$    | <code>\Im</code>    | $\angle$    | <code>\angle</code>    | $\partial$   | <code>\partial</code>   | $\spadesuit$   | <code>\spadesuit</code>   |
| $\vdash$ | <code>\vdash</code> |             |                        |              |                         |                |                           |

Table 3.7: Miscellaneous Symbols. (Shaded ones require `latexsym` package.)

|           |                   |                            |             |             |                              |           |           |                               |
|-----------|-------------------|----------------------------|-------------|-------------|------------------------------|-----------|-----------|-------------------------------|
| $\sum$    | $\sum_{i=1}^n$    | $\backslash \text{sum}$    | $\bigcap$   | $\bigcap$   | $\backslash \text{bigcap}$   | $\odot$   | $\odot$   | $\backslash \text{bigodot}$   |
| $\prod$   | $\prod_{i=1}^n$   | $\backslash \text{prod}$   | $\bigcup$   | $\bigcup$   | $\backslash \text{bigcup}$   | $\otimes$ | $\otimes$ | $\backslash \text{bigotimes}$ |
| $\coprod$ | $\coprod_{i=1}^n$ | $\backslash \text{coprod}$ | $\bigsqcup$ | $\bigsqcup$ | $\backslash \text{bigsqcup}$ | $\oplus$  | $\oplus$  | $\backslash \text{bigoplus}$  |
| $\int$    | $\int_0^1 f$      | $\backslash \text{int}$    | $\bigvee$   | $\bigvee$   | $\backslash \text{bigvee}$   | $\uplus$  | $\uplus$  | $\backslash \text{biguplus}$  |
| $\oint$   | $\oint$           | $\backslash \text{oint}$   | $\wedge$    | $\wedge$    | $\backslash \text{bigwedge}$ |           |           |                               |

Table 3.8: Variable-sized Symbols.

Here's how they look when displayed:

$$\sum_{i=1}^n x_i = \int_0^1 f$$

and in the text:  $\sum_{i=1}^n x_i = \int_0^1 f$ .

Here's how they look when displayed:

`\[\sum_{i=1}^n x_i = \int_0^1 f\]`  
and in the text:  
`\(\sum_{i=1}^n x_i = \int_0^1 f\)`.

Section 3.3.8 tells how to coerce TeX into producing  $\sum_{i=1}^n$  in a displayed formula and  $\sum_{i=1}^n$  in an in-text formula.

### Log-like Functions

In a formula like  $\log(x + y)$ , the “log”, which represents the logarithm function, is a single word that is usually set in roman type. However, typing `log` in a formula denotes the product of the three quantities  $l$ ,  $o$ , and  $g$ , which is printed as *log*. The logarithm function is denoted by the `\log` command.

Logarithms obey the law:  $\log xy = \log x + \log y$ .  $\dots \backslash ( \log xy = \log x + \log y \backslash )$ .

Other commands like `\log` for generating function names are listed in Table 3.9. Two additional commands produce the “mod” (modulo) function: `\bmod` for a binary relation and `\pmod` for a parenthesized expression. (Remember `b` as in *binary* and `p` as in *parenthesized*.)

|                      |                    |                   |                   |                      |                      |                   |                    |
|----------------------|--------------------|-------------------|-------------------|----------------------|----------------------|-------------------|--------------------|
| <code>\arccos</code> | <code>\cos</code>  | <code>\csc</code> | <code>\exp</code> | <code>\ker</code>    | <code>\limsup</code> | <code>\min</code> | <code>\sinh</code> |
| <code>\arcsin</code> | <code>\cosh</code> | <code>\deg</code> | <code>\gcd</code> | <code>\lg</code>     | <code>\ln</code>     | <code>\Pr</code>  | <code>\sup</code>  |
| <code>\arctan</code> | <code>\cot</code>  | <code>\det</code> | <code>\hom</code> | <code>\lim</code>    | <code>\log</code>    | <code>\sec</code> | <code>\tan</code>  |
| <code>\arg</code>    | <code>\coth</code> | <code>\dim</code> | <code>\inf</code> | <code>\liminf</code> | <code>\max</code>    | <code>\sin</code> | <code>\tanh</code> |

Table 3.9: Log-like Functions.

$$\gcd(m, n) = a \bmod b$$

$$x \equiv y \pmod{a+b}$$

```
\(\gcd(m, n) = a \bmod b\)
\(\equiv x \bmod y \pmod{a+b}\)
```

Note that `\pmod` has an argument and produces parentheses, while `\bmod` produces only the “mod”.

- Some log-like functions act the same as the variable-sized symbols of Table 3.8 with respect to subscripts.

As a displayed formula:

$$\lim_{n \rightarrow \infty} x = 0$$

but in text:  $\lim_{n \rightarrow \infty} x = 0$ .

As a displayed formula:

```
\[\lim_{n \rightarrow \infty} x = 0 \]
but in text:
\(\lim_{...} x = 0\).
```

### 3.3.3 Arrays

#### The array Environment

Arrays are produced with the `array` environment. It has a single argument that specifies the number of columns and the alignment of items within the columns. For each column in the array, there is a single letter in the argument that specifies how items in the column should be positioned: `c` for centered, `l` for flush left, or `r` for flush right. Within the body of the environment, adjacent rows are separated by a `\backslash` command and adjacent items within a row are separated by an `&` character.

$\begin{array}{llll}
 a+b+c & uv & x-y & 27 \\
 a+b & u+v & z & 134 \\
 a & 3u+vw & xyz & 2,978
 \end{array}$

```
\begin{array}{llll}
a+b+c & uv & x-y & 27 \\
a+b & u+v & z & 134 \\
a & 3u+vw & xyz & 2,978
\end{array}
```

There must be no `&` after the last item in a row and no `\backslash` after the last row. `TeX` is in math mode when processing each array element, so it ignores spaces. Don’t put any extra space in the argument.

In mathematical formulas, array columns are usually centered. However, a column of numbers often looks best flush right. Section 3.3.4 describes how to put large parentheses or vertical lines around an array to make a matrix or determinant.

Each item in an array is a separate formula, just as if it were in its own `math` environment. A declaration that appears in an item is local to the item; its scope is ended by the `&`, `\backslash`, or `\end{array}` that ends the item. The `\backslash` command is fragile.

### Vertical Alignment

$\text{\TeX}$  draws an imaginary horizontal center line through every formula, at the height where a minus sign at the beginning of the formula would go. An individual array item is itself a formula with a center line. The items in a row of an array are positioned vertically so their center lines are all at the same height.

Normally, the center line of an array lies where you would expect it, halfway between the top and bottom. You can change the position of an array's center line by giving an optional one-letter argument to the `array` environment: the argument `t` makes it line up with the top row's center line, while `b` makes it line up with the bottom row's center line.

The box around each array in the following formula is for clarity; it is not produced by the input:

$$x - \begin{array}{|c|} \hline a_1 \\ \vdots \\ a_n \\ \hline \end{array} - \begin{array}{|c|c|} \hline u - v & 13 \\ \hline u + v & \begin{array}{|c|} \hline 12 \\ -345 \\ \hline \end{array} \\ \hline \end{array}$$

```
... \[x - \begin{array}{c} a_{1} \\ \vdots \\ a_{n} \end{array} \\ - \begin{array}{t}[c1] \\ u-v & 13 \\ u+v & \begin{array}{b}[r] \\ 12 \\ -345 \end{array} \end{array} \] \\ \end{array}
```

### More Complex Arrays

Visual formatting is sometimes necessary to get an array to look right. Section C.1.6 explains how to change the vertical space between two rows; Sections 3.3.7 and 6.4.2 describe commands for adding horizontal space within an item; and Section C.10.2 tells how to add horizontal space between columns. The `array` environment has a number of additional features for making more complex arrays; they are described in Section C.10.2. The *L<sup>A</sup>T<sub>E</sub>X Companion* describes packages that provide additional features for the `array` environment.

The `array` environment can be used only in math mode and is meant for arrays of formulas; Section 3.6.2 describes an analogous `tabular` environment for making arrays of ordinary text items. The `array` environment is almost always used in a displayed formula, but it can appear in an in-text formula as well.

#### 3.3.4 Delimiters

A *delimiter* is a symbol that acts logically like a parenthesis, with a pair of delimiters enclosing an expression. Table 3.10 lists every symbol that  $\text{\TeX}$  regards as a delimiter, together with the command or input character that produces it. These commands and characters produce delimiters of the indicated size. However, delimiters in formulas should be big enough to “fit around” the expressions

|         |            |                                                       |
|---------|------------|-------------------------------------------------------|
| (       | )          | $\uparrow \uparrow \uparrow \uparrow$                 |
| [       | ]          | $\downarrow \downarrow \downarrow \downarrow$         |
| {       | }          | $\updownarrow \updownarrow \updownarrow \updownarrow$ |
| \lfloor | \rfloor    | $\uparrow \uparrow \uparrow \uparrow$                 |
| \lceil  | \rceil     | $\downarrow \downarrow \downarrow \downarrow$         |
| \langle | \rangle    | $\updownarrow \updownarrow \updownarrow \updownarrow$ |
| /       | \backslash |                                                       |
|         |            |                                                       |

Table 3.10: Delimiters.

that they delimit. To make a delimiter the right size, type a `\left` or `\right` command before it.

Big delimiters are most often used with arrays.

$$\left( \begin{array}{cc} x_{11} & x_{12} \\ x_{21} & x_{22} \\ y & \\ z & \end{array} \right)$$

```
... \[\left(\begin{array}{c} x_{11} x_{12} \\ x_{21} x_{22} \\ y \\ z \end{array} \right)
```

The `\left` and `\right` commands must come in matching pairs, but the matching delimiters need not be the same.

$$\vec{x} + \vec{y} + \vec{z} = \left( \begin{array}{c} a \\ b \end{array} \right)$$

```
\[... = \left(\begin{array}{c} a \\ b \end{array} \right)
```

Some formulas require a big left delimiter with no matching right one, or vice versa. The `\left` and `\right` commands must match, but you can make an invisible delimiter by typing a “.” after the `\left` or `\right` command.

$$x = \begin{cases} y & \text{if } y > 0 \\ z + y & \text{otherwise} \end{cases}$$

```
\[x = \left\{ \begin{array}{ll} y & \& \text{if } y > 0 \\ z + y & \& \text{otherwise} \end{array} \right. \right.
```

### 3.3.5 Multiline Formulas

The `displaymath` and `equation` environments make one-line formulas. A formula is displayed across multiple lines if it is a sequence of separate formulas or is too long to fit on a single line. A sequence of equations or inequalities is displayed with the `eqnarray` environment. It is very much like a three-column `array` environment, with consecutive rows separated by `\backslash` and consecutive items

within a row separated by `&` (Section 3.3.3). However, an equation number is put on every line unless that line has a `\nonumber` command.

The middle column can be anything, not just a ‘=’.

$$x = 17y \quad (2)$$

$$y > a + b + c + d + e + f + g + h + i + j + \\ k + l + m + n + o + p \quad (3)$$

```
... \begin{eqnarray}
x & = & 17y \\
y & > & a + \dots + j + \nonumber \\
& & & & k + l + m + n + o + p \\
\end{eqnarray}
```

Section 4.2 describes how to let L<sup>A</sup>T<sub>E</sub>X handle references to equations so you don’t have to remember equation numbers.

The `eqnarray*` environment is the same as `eqnarray` except it does not generate equation numbers.

$$\begin{array}{lcl} x & \ll & y_1 + \dots + y_n \\ & \leq & z \end{array} \quad \begin{array}{l} \begin{eqnarray*} x & & \ll y_{\{1\}} + \cdots + y_{\{n\}} \\ & & & & \leq z \end{eqnarray*} \end{array}$$

A `+` or `-` that begins a formula (or certain subformulas) is assumed to be a unary operator, so typing `$-x$` produces  $-x$  and typing `$\sum -x_{\{i\}}$` produces  $\sum -x_i$ , with no space between the “`-`” and the “`x`”. If the formula is part of a larger one that is being split across lines, T<sub>E</sub>X must be told that the `+` or `-` is a binary operator. This is done by starting the formula with an invisible first term, produced by an `\mbox{}` command with a null argument.

$$\begin{array}{lcl} y & = & a + b + c + d + e + f + g + h + i + j \\ & & + k + l + m + n + o + p \end{array} \quad \begin{array}{l} \begin{eqnarray*} y & & \begin{array}{l} \begin{array}{l} a + b + c + \dots + h + i + j \\ & & \& \mbox{} + k + \dots \end{array} \end{array} \end{array} \end{array}$$

A formula can often be split across lines using a `\lefteqn` command in an `eqnarray` or `eqnarray*` environment, as indicated by the following example:

$$\begin{array}{l} w + x + y + z = \\ a + b + c + d + e + f + g + h + i + j + \\ k + l + m + n + o + p \end{array} \quad \begin{array}{l} \begin{eqnarray*} w + x + y + z & = & \\ \lefteqn{w + x + y + z} & & \\ & & a + \dots + j + \\ & & k + \dots + o + p \\ \end{eqnarray*} \end{array}$$

The `\lefteqn` command works by making T<sub>E</sub>X think that the formula it produces has zero width, so the left-most column of the `eqnarray` or `eqnarray*` environment is made suitably narrow. The indentation of the following lines can be increased by adding space (with the commands of Section 6.4.2) between the `\lefteqn` command and the `\backslash\backslash`.

Breaking a single formula across lines in this way is visual formatting, and I wish  $\text{\LaTeX}$  could do it for you. However, doing it well requires more intelligence than  $\text{\LaTeX}$  has, and doing it poorly can make the formula hard to understand, so you must do it yourself. This means that the formula may have to be reformatted if you change notation (changing the formula's length) or if you change the style of your document (changing the line length).

### 3.3.6 Putting One Thing Above Another

Symbols in a formula are sometimes placed one above another. The `array` environment is good for vertically stacking subformulas, but not smaller pieces—you wouldn't use it to put a bar over an  $x$  to form  $\bar{x}$ .  $\text{\LaTeX}$  provides special commands for doing this and some other common kinds of symbol stacking.

#### Over- and Underlining

The `\overline` command puts a horizontal line above its argument.

You can have nested overlining:  $\overline{\overline{x^2 + 1}}$ .  $\dots \backslash(\backslashoverline{\backslashoverline{x^2 + 1}}\backslash).$

There's an analogous `\underline` command for underlining that works in paragraph and LR mode as well as math mode, but it's seldom used in formulas.

The value is  $3x$ .

`\underline{The} value is \$\underline{3x}\$.`

The `\underline` command is fragile.

Horizontal braces are put above or below an expression with the `\overbrace` and `\underbrace` commands.

$\overbrace{a + b + c + d}$

`\overbrace{a + \underbrace{b + c} + d}`

In a displayed formula, a subscript or superscript puts a label on the brace.

$\underbrace{a + b + \dots + y + z}_{26}^{24}$

`\underbrace{a + \overbrace{b + \dots + y}^{24} + z}_{26}`

#### Accents

The accent commands described in Section 3.2.1 are for ordinary text and cannot be used in math mode. Accents in formulas are produced with the commands shown in Table 3.11. The letter  $a$  is used as an illustration; the accents work with any letter.

Wide versions of the `\hat` and `\tilde` accent are produced by the `\widehat` and `\widetilde` commands. These commands try to choose the appropriate-sized accent to fit over their argument, but they can't produce very wide accents.

|                       |                       |                   |                     |
|-----------------------|-----------------------|-------------------|---------------------|
| $\hat{a}$ \hat{a}     | $\acute{a}$ \acute{a} | $\bar{a}$ \bar{a} | $\dot{a}$ \dot{a}   |
| $\check{a}$ \check{a} | $\grave{a}$ \grave{a} | $\vec{a}$ \vec{a} | $\ddot{a}$ \ddot{a} |
| $\breve{a}$ \breve{a} | $\tilde{a}$ \tilde{a} |                   |                     |

Table 3.11: Math Mode Accents.

Here are two sizes of wide hat:  $\widehat{1-x} = \widehat{-y}$ .  $\dots \backslash( \widehat{1-x} = \widehat{-y} \backslash).$

The letters *i* and *j* should lose their dots when accented. The commands `\imath` and `\jmath` produce a dotless *i* and *j*, respectively.

There are no dots in  $\vec{i} + \tilde{j}$ .  $\dots \backslash( \vec{\imath} + \tilde{\jmath} \backslash).$

### Stacking Symbols

The `\stackrel` command stacks one symbol above another.

$A \xrightarrow{a'} B \xrightarrow{b'} C$   $\backslash( \stackrel{a'}{A} \stackrel{b'}{B} C \dots \backslash)$   
 $\vec{x} \stackrel{\text{def}}{=} (x_1, \dots, x_n)$   $\backslash( \vec{x} \stackrel{\text{def}}{=} \mathbf{x} \dots \backslash)$

See Section 3.3.8 for an explanation of the `\mathbf` command. The `\stackrel` command's first argument is printed in small type, like a superscript; use the `\textstyle` declaration of Section 3.3.8 to print it in regular-size type.

### 3.3.7 Spacing in Math Mode

In math mode,  $\TeX$  ignores the spaces you type and formats the formula the way it thinks is best. Some authors feel that  $\TeX$  cramps formulas, and they want to add more space.  $\TeX$  knows more about typesetting formulas than many authors do. Adding extra space usually makes a formula prettier but harder to read, because it visually “fractures” the formula into separate units. Study how formulas look in ordinary mathematics texts before trying to improve  $\TeX$ 's formatting.

Although fiddling with the spacing is dangerous, you sometimes have to do it to make a formula look just right. One reason is that  $\TeX$  may not understand the formula's logical structure. For example, it interprets  $y \, dx$  as the product of three quantities rather than as  $y$  times the differential  $dx$ , so it doesn't add the little extra space after the  $y$  that appears in  $y \, dx$ . Section 3.4 explains how to define your own commands for expressing this kind of logical structure, so you need worry about the proper spacing only when defining the commands, not when writing the formulas.

Like any computer program that makes aesthetic decisions,  $\TeX$  sometimes needs human assistance. You'll have to examine the output to see if the spacing

needs adjustment. Pay special attention to square root signs, integral signs, and quotient symbols (/).

The following five commands add the amount of horizontal space shown between the vertical lines:

|                        |                 |                    |
|------------------------|-----------------|--------------------|
| \, thin space          | \: medium space | \_ interword space |
| \! negative thin space | \; thick space  |                    |

The \! acts like a backspace, removing the same amount of space that \, adds. The \, command can be used in any mode; the others can appear only in math mode. Here are some examples of their use, where the result of omitting the spacing commands is also shown.

|                 |                            |
|-----------------|----------------------------|
| $\sqrt{2}x$     | $\sqrt{2}x$                |
| $n/\log n$      | $n/\log n$                 |
| $\iint z dx dy$ | $\iint z dx dy$            |
| $\sqrt{2}x$     | instead of $\sqrt{2}x$     |
| $n/\log n$      | instead of $n/\log n$      |
| $\iint z dx dy$ | instead of $\iint z dx dy$ |

As with all such fine tuning, you should not correct the spacing in formulas until you've finished writing the document and are preparing the final output.

### 3.3.8 Changing Style in Math Mode

#### Type Style

$\text{\LaTeX}$  provides the following commands for changing type style in math mode:

|                                            |                                                    |
|--------------------------------------------|----------------------------------------------------|
| <i>italic</i> + $2^{ft}\Psi\log[\psi]$     | $\$\\mathit{italic} + 2^{ft} \Psi \log[\psi] \$$   |
| <i>roman</i> + $2^{ft}\Psi\log[\psi]$      | $\$\\mathrm{roman} + 2^{ft} \Psi \log[\psi] \$$    |
| <b>bold</b> + $2^{ft}\Psi\log[\psi]$       | $\$\\mathbf{bold} + 2^{ft} \Psi \log[\psi] \$$     |
| <i>sans serif</i> + $2^{ft}\Psi\log[\psi]$ | $\$\\mathsf{sans\\ serif} + ... \$$                |
| <i>typewriter</i> + $2^{ft}\Psi\log[\psi]$ | $\$\\mathtt{typewriter} + ... \$$                  |
| <i>CAL UPPERCASE LETTERS ONLY</i>          | $\$\\mathcal{CAL\\ UPPERCASE\\ LETTERS\\ ONLY} \$$ |

They change the style only of letters, numbers, and uppercase Greek letters. Nothing else is affected.  $\text{\LaTeX}$  normally uses an italic type style for letters in math mode. However, it interprets each letter as a separate mathematical symbol, and it produces very different spacing than an ordinary italic typestyle. You should use  $\mathit$  for multiletter symbols in formulas.

Is *different* any *different* from  $dif^2e^2rnt?$

Is  $\$different\$$  any  $\$\\mathit{different} \$$ ...

As is evident from this example, you should not use  $\$... \$$  as a shorthand for  $\text{\emph{...}}$  or  $\text{\textit{...}}$ .

The  $\boldsymbol{\text{\boldsymbol{boldmath}}}$  declaration causes everything in a formula to be bold, including the symbols.

*diff* +  $2^{ft}\Psi\log[\psi]$

$\$\\boldmath \$\\{diff + 2^{ft} \Psi \log[\psi] \$$

This declaration cannot be used in math mode. To produce a bold subformula, put the subformula in an `\mbox`.

$x + \nabla f = 0$

`\(x + \mbox{\boldmath $\nabla f$} = 0\)`

The `\boldmath` command generates a warning if any font (size and style of type) that you *might* use in a formula isn't available. For example, it might issue a warning that L<sup>A</sup>T<sub>E</sub>X does not have the font needed to produce a bold sans serif subsubscript, even though you never dreamed of producing such a subsubscript. You should examine your output to make sure that the fonts you do use are the right ones.

### Math Style

T<sub>E</sub>X uses the following four *math styles* when typesetting formulas:

**display** For normal symbols in a displayed formula.

**text** For normal symbols in an in-text formula.

**script** For subscripts and superscripts.

**scriptscript** For further levels of sub- and superscripting, such as subscripts of superscripts.

Display and text math styles are the same except for the size of the variable-sized symbols in Table 3.8 on page 44 and the placement of subscripts and superscripts on these symbols, on some of the log-like functions in Table 3.9 on page 44, and on horizontal braces. T<sub>E</sub>X uses small type in script style and even smaller type in scriptscript style. The declarations `\displaystyle`, `\textstyle`, `\scriptstyle`, and `\scriptscriptstyle` force T<sub>E</sub>X to use the indicated style.

Compare the small superscript in  $e^{x(i)}$  with the ... small superscript in `\( e^{x(i)} \)`  
large one in  $e^{y(i)}$ . ... large one in `\( e^{\textstyle y(i)} \)`.

### 3.3.9 When All Else Fails

If you write a lot of complicated formulas, sooner or later you'll encounter one that can't be handled with the commands described so far. When this happens, the first thing to do is look at the advanced L<sup>A</sup>T<sub>E</sub>X features described in Sections C.7 and C.10.2. If you don't find what you need there, consider using the `amstex` package, described in the *L<sup>A</sup>T<sub>E</sub>X Companion*. This package was created for mathematicians by the American Mathematical Society.

If you are not a mathematician and don't write many complicated formulas, you may be satisfied with an *ad hoc* solution to your problem that uses visual formatting. Try the commands in Section 6.4.3 that move text around and change how big T<sub>E</sub>X thinks it is. L<sup>A</sup>T<sub>E</sub>X's `picture` environment, described in

Section 7.1, allows you to put a symbol anywhere you want, and to draw lines and arrows. The **graphics** package described in Section 7.2 contains commands to shrink, stretch, and rotate text. With a little effort, you can handcraft just about any kind of formula you want.

## 3.4 Defining Commands and Environments

The input file should display as clearly as possible the logical structures in your document. Any structure, such as a special mathematical notation, that appears several times in the document should be expressed with its own command or environment. The following two sections explain how to define commands and environments. Section 3.4.3 describes how to type theorems and similar structures.

### 3.4.1 Defining Commands

The simplest type of repeated structure occurs when the same text appears in different places. The **\newcommand** declaration defines a new command to generate such text; its first argument is the command name and its second argument is the text.

Let  $\Gamma_i$  be the number of gnats per cubic meter,  
where  $\Gamma_i$  is normalized with respect to  $\nu(s)$ .

**\newcommand{\gn}{\\$\\Gamma\_{i\\$}}**

...

Let **\gn** be the ... where **\gn** is ...

The **\\_** commands are needed because **TeX** ignores space characters following the command name **\gn**.

This example illustrates a common problem in defining commands to produce mathematical formulas. The **\Gamma** command can be used only in math mode, which is why the **\$**'s are needed in the **\newcommand** argument. However, the command **\gn** cannot be used in math mode because the first **\$** would cause **TeX** to leave math mode and encounter the **\Gamma** while in paragraph mode. To solve this problem, **L<sup>A</sup>T<sub>E</sub>X** provides the **\ensuremath** command, which ensures that its argument is typeset in math mode, regardless of the current mode. Thus, **\ensuremath{x^2}** is equivalent to **\$x^2\$** if it appears in LR or paragraph mode, and it is equivalent to **x<sup>2</sup>** if it appears in math mode. With **\ensuremath**, we define **\gn** so it can be used in any mode.

Let  $\Gamma_i$  be the number of gnats per cubic meter,  
where  $e^{\Gamma_i} = \nu(s)$ .

**\newcommand{\gn}{\ensuremath{\Gamma\_i}}**

...

Let **\gn** be the ... where **\$e^{\gn}=\nu(s)\$**.

You should use **\ensuremath** when defining a command to be used both in and out of math mode.

In addition to making the input more readable, defining your own commands can save typing. L<sup>A</sup>T<sub>E</sub>X's command and environment names have generally been chosen to be descriptive rather than brief. You can use `\newcommand` to define abbreviations for frequently used commands. For example, the declarations

```
\newcommand{\be}{\begin{enumerate}}
\newcommand{\ee}{\end{enumerate}}
```

define `\be` ... `\ee` to be equivalent to

```
\begin{enumerate} ... \end{enumerate}
```

For repetitive structures having components that vary, you can define commands with arguments by giving `\newcommand` an optional argument.

Since  $gnu(5x; y)$  and  $gnu(5x-1; y+1)$  represent adjacent populations, they are approximately equal.

```
\newcommand{\gnaw}[2]{\ensuremath{\mathit{gnu}(\#1;\#2)}}
```

Since `\gnaw{5x}{y}` and `\gnaw{5x-1}{y+1}` ...

The optional argument 2 (in square brackets) specifies that `\gnaw` has two arguments. The `#1` and `#2` in the last argument of `\newcommand` are called *parameters*; they are replaced by the first and second arguments, respectively, of `\gnaw` when that command is used. A command may have up to nine arguments. Section C.8.1 explains how to define a command that has an optional argument.

When you define a command like `\gnaw`, the definition is saved exactly as it appears in the `\newcommand` declaration. When T<sub>E</sub>X encounters a use of `\gnaw`, it replaces `\gnaw` and its arguments by the definition, with the arguments substituted for the corresponding *parameters*—the `#1` replaced by the first argument and the `#2` replaced by the second. T<sub>E</sub>X then processes this text pretty much as if you had typed it instead of typing the `\gnaw` command. However, defining a command to have space at the end is usually a bad idea, since it can lead to extra space in the output when the command is used.

One command can be defined in terms of another.

The definition above of  $gnu(0; 1)$  gives  $gnu(5x; y)$  the expected value.

```
\newcommand{\usegnaw}{\gnaw{5x}{y}}
... of \gnaw{0}{1} gives \usegnaw\ the ...
```

It doesn't matter whether the `\newcommand` declaration defining `\usegnaw` comes before or after the one defining `\gnaw`, so long as they both come before any use of `\usegnaw`. However, a command cannot be defined in terms of itself, since T<sub>E</sub>X would chase its tail forever trying to figure out what such a definition meant.<sup>2</sup>

When T<sub>E</sub>X encounters a command, it looks for that command's arguments before interpreting it or any subsequent commands. Thus, you can't type

<sup>2</sup>This kind of recursive definition is possible using more advanced T<sub>E</sub>X commands, but it cannot be done with the L<sup>A</sup>T<sub>E</sub>X commands described in this book.

---

```
\newcommand{\gnawargs}{{5x}{y}} \gnaw\gnawargs is wrong
```

because TeX expects the `\gnaw` command to be followed by two arguments enclosed in braces, not by another command.

The braces surrounding the last argument of the `\newcommand` declaration do not become part of the command's definition, and the braces surrounding an argument are thrown away before substituting the argument for the corresponding parameter. This means that the braces delimiting an argument do not delimit the scope of declarations in the argument. To limit the scope of declarations contained within an argument, you must add explicit braces in the command definition.

*gnus(x;54)* is fine, but in *gnus(x;54)*, the scope of the *emphasis* declaration extends into the following text.

```
\newcommand{\good}[3]{\#1$({\#2};{\#3})$}
\newcommand{\bad}[3]{\#1$({\#2};{\#3})$}
...
\good{\em gnus}{x}{54} is fine, but in
\bad{\em gnus}{x}{54}, the scope ...
```

Using `\newcommand` to define a command that already exists produces an error. The `\renewcommand` declaration redefines an already defined command; it has the same arguments as `\newcommand`. Don't redefine an existing command unless you know what you're doing. Even if you don't explicitly use a command, redefining it can produce strange and unpleasant results. Also, never define or redefine any command whose name begins with `\end`.

The `\newcommand` and `\renewcommand` commands are declarations; their scopes are determined by the rules given in Section 2.2.1. It's best to put all command definitions together in the preamble, with comments describing what the commands do. You're likely to re-use many of them in subsequent documents.

### 3.4.2 Defining Environments

The `\newenvironment` command is used to define a new environment. A command of the form

```
\newenvironment{cozy}{begin text}{end text}
```

defines a *cozy* environment for which TeX replaces a `\begin{cozy}` command by the *begin text* and an `\end{cozy}` command by the *end text*. A new environment is usually defined in terms of an existing environment such as `itemize`, with the *begin text* beginning the `itemize` environment and the *end text* ending it.

Here is an example of a user-defined environment:

- *This environment produces items that are emphasized.*
- *It is defined in terms of L<sup>A</sup>T<sub>E</sub>X's `itemize` environment and `\em` command.*

```
\newenvironment{emphit}{\begin{itemize}
 \em}{\end{itemize}}
... example of a user-defined environment:
\begin{emphit}
\item This environment produces ...
\end{emphit}
```

An optional argument of the `\newenvironment` command allows you to define an environment that has arguments; it works the same as described above for `\newcommand`.

Observe how a new logical structure—in this example, a labeled description of a single item—can be defined in terms of existing environments.

*Armadillos:* This witty description of the armadillo was produced by the `descit` environment.

```
\newenvironment{descit}[1]{\begin{quote}
 \textit{\#1}:}{\end{quote}}
...
defined in terms of existing environments.
\begin{descit}{Armadillos}
This witty description of the armadillo ...
\end{descit}
```

The parameters (the #1, #2, etc.) can appear only in the *begin text*. The comments made above about the scope of declarations that appear inside arguments of a command defined with `\newcommand` apply to the arguments of environments defined with `\newenvironment`. Section C.8.2 explains how to define an environment with an optional argument.

The `\newenvironment` command produces an error if the environment is already defined. Use `\renewenvironment` to redefine an existing environment. If `\newenvironment` complains that an environment you've never heard of already exists, choose a different environment name. Use `\renewenvironment` only when you know what you're doing; don't try redefining an environment that you don't know about.

### 3.4.3 Theorems and Such

Mathematical text usually includes theorems and/or theorem-like structures such as lemmas, propositions, axioms, conjectures, and so on. Nonmathematical text may contain similar structures: rules, laws, assumptions, principles, etc. Having a built-in environment for each possibility is out of the question, so L<sup>A</sup>T<sub>E</sub>X provides the `\newtheorem` declaration to define environments for the particular theorem-like structures in your document.

The `\newtheorem` command has two arguments: the first is the name of the environment, the second is the text used to label it.

Conjectures are numbered consecutively from the beginning of the document; this is the fourth one:

**Conjecture 4** *All conjectures are interesting, but some conjectures are more interesting than others.*

```
\newtheorem{guess}{Conjecture}
...
document; this is the fourth one:
\begin{guess}
 All conjectures ... than others.
\end{guess}
```

The `\newtheorem` declaration is best put in the preamble, but it can go anywhere in the document.

A final optional argument to `\newtheorem` causes the theorem-like environment to be numbered within the specified sectional unit.

This is the first Axiom of Chapter 3:

**Axiom 3.1** *All axioms are very dull.*

```
\newtheorem{axiom}{Axiom}[chapter]
...
\begin{axiom}
 All axioms are very dull.
\end{axiom}
```

Theorem-like environments can be numbered within any sectional unit; using `section` instead of `chapter` in the example above causes axioms to be numbered within sections.

Sometimes one wants different theorem-like structures to share the same numbering sequence—so, for example, the hunch immediately following Conjecture 5 should be Hunch 6.

**Conjecture 5** *Some good conjectures are numbered.*

**Hunch 6** *There are no sure-fire hunches.*

```
\newtheorem{guess}{Conjecture}
\newtheorem{hunch}{guess}[Hunch]
...
\begin{guess} Some good ... \end{guess}
\begin{hunch} There are ... \end{hunch}
```

The optional argument `guess` in the second `\newtheorem` command specifies that the `hunch` environment should be numbered in the same sequence as the `guess` environment.

A theorem-like environment defined with `\newtheorem` has an optional argument that is often used for the inventor or common name of a theorem, definition, or axiom.

**Conjecture 7 (Wiles, 1985)** *There do exist integers  $n > 2$ ,  $x$ ,  $y$ , and  $z$  such that  $x^n + y^n = z^n$ .*

```
\begin{guess}[Wiles, 1985]
 There do exist integers $n>2$, x, ...
\end{guess}
```

See Section C.1.1 if the body of a theorem-like environment begins with a `[`.

## 3.5 Figures and Other Floating Bodies

### 3.5.1 Figures and Tables

TeX will break a sentence across pages to avoid a partially filled page. But some things, such as pictures, cannot be split; they must be “floated” to convenient places, like the top of the following page, to prevent half-empty pages. LATeX provides two environments that cause their contents to float in this way: **figure** and **table**. There are packages that define or allow you to define other environments for floating objects, such as a **program** environment for computer programs. LATeX doesn’t care what you use any of these environments for; so far as it’s concerned, the only difference between them is how they are captioned.

The caption on a figure or table is made with a `\caption` command having the caption’s text as its argument. This is a moving argument, so fragile commands must be `\protect`’ed (see Section 2.2.3). The **figure** or **table** environment is placed in with the text, usually just past the point where the figure or table is first mentioned.

The body of the figure goes here. This figure happened to float to the top of the current page.

Figure 7: The caption goes here.

⋮

This is the place in the running text that mentions Figure 7 for the first time. The figure will not be put on an earlier page than the text preceding the **figure** environment.

This is the place in the running text that mentions Figure 7 for the first time.

```
\begin{figure}
 The body of the figure goes here.
 This figure ... the current page.
 \caption{The caption goes here.}
\end{figure}
```

The figure will not be put on an ...

Tables are numbered separately from figures, using the same numbering scheme. Section 4.2 explains how to number cross-references automatically, so you never have to type the actual figure numbers.

You can put anything you want in the body of a figure or table; LATeX processes it in paragraph mode just like any other text. The **figure** environment is generally used for pictures and the **table** environment for tabular information. Simple pictures can be drawn with the **picture** environment of Section 7.1. You can insert pictures prepared with other programs using the **graphics** package, described in Section 7.2. You can also use the `\vspace` command of Section 6.4.2 to leave room for a picture to be pasted in later. Tabular material can be formatted with the **tabular** environment of Section 3.6.2. Section 6.5 explains how to center the figure or table.

The body of a figure or table is typeset as a paragraph the same width as in the ordinary running text. Section 6.4.3 explains how to make paragraphs of different widths, position two half-width figures side by side, and do other

formatting within a `figure` or `table` environment. More than one `\caption` command can appear in the same `figure` or `table` environment, producing a single floating object with multiple numbered captions. The `\caption` command can be used only in a `figure` or `table` environment.

`LATEX` ordinarily tries to place a figure or table either above the text at the top of a page, below the text at the bottom of a page, or on a separate page containing nothing but figures and tables. Section C.9.1 describes the rules by which `LATEX` decides where a floating object should float and how you can influence its decision; read that section if you don't understand why `LATEX` put a figure or table where it did.

### 3.5.2 Marginal Notes

A marginal note is made with the `\marginpar` command, having the text as its argument. The note is placed in the margin, its first line even with the line of text containing the command. `TeX` is in paragraph mode when processing the marginal note. The following example shows how I typed this paragraph:

*This is a marginal note.*

and, having the text as its line even with the line of mode when processing the yped this paragraph.

... placed in the margin,  
`\marginpar{\em This is a marginal note.}`  
 its first line even with the line of  
 ... how I typed this paragraph.

The standard styles put notes in the right margin for one-sided printing (the default), in the outside margin for two-sided printing (specified by the `twoside` document-class option), and in the nearest margin for two-column formatting (the `twocolumn` option). Section C.9.2 describes commands for getting `LATEX` to put them in the opposite margin.

You may want a marginal note to vary depending upon which margin it's in. For example, to make an arrow pointing to the text, you need a left-pointing arrow in the right margin and a right-pointing one in the left margin. If the `\marginpar` command is given an optional first argument, it uses that argument if the note goes in the left margin and uses the second (mandatory) argument if the note goes in the right margin. The command

```
\marginpar [\$\Rightarrow\$]{\$\Leftarrow\$}
```

makes an arrow that points towards the text, whichever margin the note appears in.<sup>3</sup>

A marginal note is never broken across pages; a note that's too long will extend below the page's bottom margin. `LATEX` moves a marginal note down on the page to keep it from bumping into a previous one, warning you when it does

<sup>3</sup>The arrows won't be symmetrically placed, since both will be at the left of the space reserved for marginal notes. The `\hfill` command of Section 6.4.2 can be used to adjust their horizontal position.

so. When using notes more than two or three lines long, you may have to adjust their placement according to where they happen to fall on the page. The vertical position of a note is changed by beginning it with a vertical spacing command (Section 6.4.2). You may also have to use the commands of Section 6.2.2 to control where L<sup>A</sup>T<sub>E</sub>X starts a new page. This is visual design, which means reformatting if you make changes to the document. Save this job until the very end, after you've finished all the writing.

Marginal notes are not handled efficiently by L<sup>A</sup>T<sub>E</sub>X; it may run out of space if you use too many of them. How many you can use before this happens depends upon what computer you're running L<sup>A</sup>T<sub>E</sub>X on and how many figures and tables you have.

## 3.6 Lining It Up in Columns

The **tabbing** and **tabular** environments both can align text in columns. The **tabbing** environment allows you to set tab stops similar to the ones on a typewriter, while the **tabular** environment is similar to the **array** environment described in Section 3.3.3, except that it is for ordinary text rather than formulas. The **tabbing** and **tabular** environments differ in the following ways:

- The **tabbing** environment can be used only in paragraph mode and makes a separate paragraph. The **tabular** environment can be used in any mode; it can put a table in the middle of a formula or line of text.
- T<sub>E</sub>X can start a new page in the middle of a **tabbing** environment, but not in the middle of a **tabular** environment. Thus, a long **tabbing** environment can appear in the running text, but a long **tabular** environment should go in a figure or table (Section 3.5.1).
- T<sub>E</sub>X automatically determines the widths of columns in the **tabular** environment; you have to do that yourself in the **tabbing** environment by setting tab stops.
- A change of format in the middle of the environment is easier in the **tabbing** than in the **tabular** environment. This makes the **tabbing** environment better at formatting computer programs.

If neither the **tabbing** nor the **tabular** environment does what you want, consult the L<sup>A</sup>T<sub>E</sub>X *Companion*. It describes several packages that provide very powerful commands to format tabular material.

### 3.6.1 The **tabbing** Environment

In the **tabbing** environment, you align text in columns by setting tab stops and tabbing to them, somewhat as you would with an ordinary typewriter. Tab



A lively *gnat*A dull *gnu*

A lively \em gnat \&gt; A dull gnu \\

The **tabbing** environment has a number of additional features that are described in Section C.10.1.

### 3.6.2 The tabular Environment

The **tabular** environment is similar to the **array** environment, so you should read Section 3.3.3 before reading any further here. It differs from the **array** environment in two ways: it may be used in any mode, not just in math mode, and its items are processed in LR mode instead of in math mode. This makes **tabular** better for tabular lists and **array** better for mathematical formulas. This section describes some features used mainly with the **tabular** environment, although they apply to **array** as well. Many additional features of these environments are described in Section C.10.2.

A **|** in the **tabular** environment's argument puts a vertical line extending the full height of the environment in the specified place. An **\hline** command after a **\\\** or at the beginning of the environment draws a horizontal line across the full width of the environment. The **\cline{i-j}** command draws a horizontal line across columns *i* through *j*, inclusive.

|           |         |         |
|-----------|---------|---------|
| gnats     | gram    | \$13.65 |
|           | each    | .01     |
| gnu       | stuffed | 92.50   |
| emu       |         | 33.33   |
| armadillo | frozen  | 8.99    |

```
\begin{tabular}{||l|l|r||}
\hline
gnats & gram & $13.65 \\
& each & .01 \\
\hline
gnu & stuffed & 92.50 \\
& \cline{1-1} \cline{3-3} \\
emu & & 33.33 \\
armadillo & frozen & 8.99 \\
\hline
\end{tabular}
```

This is the only situation in which a **\\\** goes after the last row of the environment.

Single (not doubled) **|** specifiers in the argument of a **tabular** (or **array**) environment do not change the width of the table. Tables not enclosed by vertical lines therefore have extra space around them. Section C.10.2 explains how to remove this space.

A single item that spans multiple columns is made with a **\multicolumn** command, having the form

```
\multicolumn{n}{pos}{item}
```

where *n* is the number of columns to be spanned, *pos* specifies the horizontal positioning of the item—just as in the environment's argument—and *item* is the item's text. The *pos* argument replaces the portion of the environment's argument corresponding to the *n* spanned columns; it must contain a single **l**, **r**, or **c** character and may contain **|** characters.

Note the placement of “ITEM” and “PRICE”:

| ITEM         | PRICE     |
|--------------|-----------|
| gnat (dozen) | 3.24      |
| gnu (each)   | 24,985.47 |

```
... \begin{tabular}{llr}
 \multicolumn{2}{c}{ITEM} &
 \multicolumn{1}{c}{PRICE} \\
 gnat & (dozen) & 3.24 \\
 gnu & (each) & 24,985.47
\end{tabular}
```

A `\multicolumn` command spanning a single column serves to override the item positioning specified by the environment argument.

When the environment argument has `|` characters, it’s not obvious which of them get replaced by a `\multicolumn`’s positioning argument. The rule is: *the part of the environment argument corresponding to any column other than the first begins with an `l`, `r`, or `c` character*. By this rule, the argument `|l|l|r|` is split into parts as `|l|, |l|, |r|`.

| <i>type</i>  | <i>style</i> |       |
|--------------|--------------|-------|
| smart        | red          | short |
| rather silly | puce         | tall  |

```
\begin{tabular}{|l|l|r|} \hline \hline
\emph{type} & \multicolumn{2}{c}{\emph{style}} \\ \hline
smart & red & short \\ \hline
rather silly & puce & tall \\ \hline \hline
\end{tabular}
```

The `tabular` environment produces an object that `TeX` treats exactly like a single, big letter. You could put it in the middle of a paragraph, or in the middle of a word—but that would look rather strange. A `tabular` environment is usually put in a figure or table (Section 3.5.1), or else displayed on a line by itself, using the `center` environment of Section 6.5.

The `tabular` environment makes it easy to draw tables with horizontal and vertical lines. Don’t abuse this feature. Lines usually just add clutter to a table; they are seldom helpful. The numbered tables in this book do not contain a single line.

## 3.7 Simulating Typed Text

A printed document may contain simulated typed text—for example, the instruction manual for a computer program usually shows what the user types. The `\ttfamily` declaration causes `TeX` to produce text in a typewriter type style (Section 3.1), but it doesn’t stop `TeX` from breaking the text into lines as it sees fit. The `verbatim` environment allows you to type the text exactly the way you want it to appear in the document.

The **verbatim** environment is the one place where L<sup>A</sup>T<sub>E</sub>X pays attention to how the input file is formatted.

What the `#$\^~` is ‘‘going’’ {on}  
here `\today \\\??`

... to how the input file is formatted.

```
\begin{verbatim}
What the #\$^~ is "going" {on}
here \today \\\??
\end{verbatim}
```

Each space you type produces a space in the output, and new lines are begun just where you type them. Special characters such as `\` and `{` are treated like ordinary characters in a **verbatim** environment. In fact, you can type anything in the body of a **verbatim** environment except for the fourteen-character sequence `\end{verbatim}`.

The **verbatim** environment begins on a new line of output, as does the text following it. A blank line after the `\end{verbatim}` starts a new paragraph as usual.

The `\verb` command simulates a short piece of typed text inside an ordinary paragraph. Its argument is not enclosed in braces, but by a pair of identical characters.

The `\verb+}` {gnat and `--#` gnus are silly.

The `\verb+}` {gnat and `\verb2--#2` ...

The argument of the first `\verb` command is contained between the two `+` characters, the argument of the second between two `2` characters. Instead of `+` or `2`, you can use any character that does not appear in the argument except a space, a letter, or a `*`.

There are also a **verbatim\*** environment and a `\verb*` command. They are exactly like **verbatim** and `\verb` except that a space produces a `_` symbol instead of a blank space.

You can type `$x_=y$` or `_x=y$_`.

... `\verb*|$x = y$|` or `\verb*/ $x=y$ /`.

The **verbatim** environment and `\verb` command are inherently anomalous, since characters like `$` and `}` don't have their usual meanings. This results in the following restrictions on their use:

- A **verbatim** environment or `\verb` command may not appear within an argument of any other command. (However, they may appear inside another environment.)
- There may be no space between a `\verb` or `\verb*` command and its argument. The command and its argument must all appear on a single line of the input file.
- There may be no space between `\end` and `{` in `\end{verbatim}`.

The **verbatim** environment is for simulating typed text; it is not intended to turn L<sup>A</sup>T<sub>E</sub>X into a typewriter. If you're tempted to use it for visual formatting, don't; use the **tabbing** environment of Section 3.6.1 instead.

## CHAPTER 4

# Moving Information Around



*L**A**T*<sub>E</sub>*X* often has to move information from one place to another. For example, the information contained in a table of contents comes from the sectioning commands that are scattered throughout the input file. Similarly, the *L**A**T*<sub>E</sub>*X* command that generates a cross-reference to an equation must get the equation number from the `equation` environment, which may occur several sections later. This chapter describes the features that cause *L**A**T*<sub>E</sub>*X* to move information around. You also move information around when you send your document to friends and colleagues. Section 4.7 describes commands that make this easier.

*L**A**T*<sub>E</sub>*X* requires two passes over the input to move information around: one pass to find the information and a second pass to put it into the text. (It occasionally even requires a third pass.) To compile a table of contents, one pass determines the titles and starting pages of all the sections and a second pass puts this information into the table of contents. Instead of making two passes every time it is run, *L**A**T*<sub>E</sub>*X* reads your input file only once and saves the cross-referencing information in special files for use the next time. For example, if `sample2e.tex` had a command to produce a table of contents, then *L**A**T*<sub>E</sub>*X* would write the necessary information into the file `sample2e.toc`. It would use the information in the current version of `sample2e.toc` to produce the table of contents, and would write a new version of that file to produce the table of contents the next time *L**A**T*<sub>E</sub>*X* is run with `sample2e.tex` as input.

*L**A**T*<sub>E</sub>*X*'s cross-referencing information is therefore always “old”, since it was gathered on a previous execution. This will be noticeable mainly when you are first writing the document—for example, a newly added section won't be listed in the table of contents. However, the last changes you make to your document will normally be minor ones that polish the prose rather than add new sections or equations. The cross-referencing information is unlikely to change the last few times you run *L**A**T*<sub>E</sub>*X* on your file, so all the cross-references will almost always be correct in the final version. In any case, if the cross-referencing is incorrect, *L**A**T*<sub>E</sub>*X* will type a warning message when it has finished. (But, *L**A**T*<sub>E</sub>*X* won't warn you about changes to table-of-contents entries.) Running it again on the same input will correct any errors.<sup>1</sup>

An error in your input file could produce an error in one of the special cross-referencing files. The error in the cross-referencing file will not manifest itself until that file is read, the next time you run *L**A**T*<sub>E</sub>*X*. Section 8.1 explains how to recognize such an error.

## 4.1 The Table of Contents

A `\tableofcontents` command produces a table of contents. More precisely, it does two things:

---

<sup>1</sup>If you're a computer wizard or are very good at mathematical puzzles, you may be able to create a file in which a reference to a page number always remains incorrect. The chance of that happening by accident is small.

- It causes L<sup>A</sup>T<sub>E</sub>X to write a new `toc` file—that is, a file with the same first name as the input file and the extension `toc`—with the information needed to generate a table of contents.
- It reads the information from the previous version of the `toc` file to produce a table of contents, complete with heading.

The commands `\listoffigures` and `\listoftables` produce a list of figures and a list of tables, respectively. They work just like the `\tableofcontents` command, except that L<sup>A</sup>T<sub>E</sub>X writes a file with extension `lof` when making a list of figures and a file with extension `lot` when making a list of tables.

Occasionally, you may find that you don't like the way L<sup>A</sup>T<sub>E</sub>X prints a table of contents or a list of figures or tables. You can fine-tune an individual entry by using the optional argument to the sectioning command or `\caption` command that generates it; see Sections C.4.1 and C.9.1. Formatting commands can also be introduced with the `\addtocontents` command described in Section C.4.3. If all else fails, you can edit the `toc`, `lof`, and `lot` files yourself. Edit these files only when preparing the final version of your document, and use a `\nofiles` command (described in Section C.11.1) to suppress the writing of new versions of the files.

## 4.2 Cross-References

One reason for numbering things like figures and equations is to refer the reader to them, as in: “See Figure 3 for more details.” You don't want the “3” to appear in the input file because adding another figure might make this one become Figure 4. Instead, you can assign a *key* of your choice to the figure and refer to it by that key, letting L<sup>A</sup>T<sub>E</sub>X translate the reference into the figure number. The key is assigned a number by the `\label` command, and the number is printed by the `\ref` command. A `\label` command appearing in ordinary text assigns to the key the number of the current sectional unit; one appearing inside a numbered environment assigns that number to the key.

Equation 4.12 in Section 2.3 below is Euler's famous result.

⋮

### 2.3 Early Results

Euler's equation

$$e^{i\pi} + 1 = 0 \quad (4.12)$$

combines the five most important numbers in mathematics in a single equation.

```
Equation^{\ref{eq:euler}} in
Section^{\ref{sec-early}} below
...
\subsection{Early Results}
\label{sec-early}
Euler's equation
\begin{equation}
e^{i\pi} + 1 = 0 \quad \label{eq:euler}
\end{equation}
\end{equation}
combines the five most important ...
```

In this example, the `\label{eq:euler}` command assigns the key `eq:euler` to the equation number, and the command `\ref{eq:euler}` generates that equation number.

A key can consist of any sequence of letters, digits, or punctuation characters (Section 2.1). Upper- and lowercase letters are different, so `gnu` and `Gnu` are distinct keys. In addition to sectioning commands, the following environments also generate numbers that can be assigned to keys with a `\label` command: `equation`, `eqnarray`, `enumerate` (assigns the current item's number), `figure`, `table`, and any theorem-like environment defined with the `\newtheorem` command of Section 3.4.3.

The `\label` command can usually go in any natural place. To assign the number of a sectional unit to a key, you can put the `\label` command anywhere within the unit except within a command argument or environment in which it would assign some other number, or you can put it in the argument of the sectioning command. To refer to a particular equation in an `eqnarray` environment, put the `\label` command anywhere between the `\begin{eqnarray}` that begins the equation and the `\end{eqnarray}` that ends it. The position of the `\label` command in a figure or table is less obvious: it must go after the `\caption` command or in its argument.

Figure 17 shows the evolution of the salamander (order Urodela) from its origin in the Jurassic ...

Body of Figure

Figure 17: Newts on parade.

Figure~\ref{fig:newt} shows the evolution  
`\begin{figure}`  
`\centering` Body of Figure  
`\caption{Newts on parade.}` `\label{fig:newt}`  
`\end{figure}`  
of the salamander...

(See Section 2.2.1 for an explanation of the `\~` command.) A `\caption` command within its `figure` or `table` environment acts like a sectioning command within the document. Just as a document has multiple sections, a figure or table can have multiple captions.

The `\pageref` command is similar to the `\ref` command except it produces the page number of the place in the text where the corresponding `\label` command appears.

See page 42 for more details.

...

*Text on page 42:*

The meaning of life, the universe, and ...

See page~\pageref{'meaning'} for more

...

The `\label{'meaning'}` meaning of life, ...

A `\ref` or `\pageref` command generates only the number, so you have to type the page to produce “page 42”.

The numbers generated by `\ref` and `\pageref` were assigned to the keys the previous time you ran L<sup>A</sup>T<sub>E</sub>X on your document. Thus, the printed output

will be incorrect if any of these numbers have changed. L<sup>A</sup>T<sub>E</sub>X will warn you if this may have happened, in which case you should run it again on the input file to make sure the cross-references are correct. (This warning will occur if any number assigned to a key by a `\label` command has changed, even if that number is not referenced.) Each `\ref` or `\pageref` referring to an unknown key produces a warning message; such messages appear the first time you process any file containing these commands.

A `\label` can appear in the argument of a sectioning or `\caption` command, but in no other moving argument.

Using keys for cross-referencing saves you from keeping track of the actual numbers, but it requires you to remember the keys. You can produce a list of the keys by running L<sup>A</sup>T<sub>E</sub>X on the input file `lablst.tex`. (You probably do this by typing “`latex lablst`”; check your *Local Guide* to be sure.) L<sup>A</sup>T<sub>E</sub>X will then ask you to type in the name of the input file whose keys you want listed, as well as the name of the document class specified by that file’s `\documentclass` command.

## 4.3 Bibliography and Citation

A citation is a cross-reference to another publication, such as a journal article, called the *source*. The modern method of citing a source is with a cross-reference to an entry in a list of sources at the end of the document. With L<sup>A</sup>T<sub>E</sub>X, the citation is produced by a `\cite` command having the citation key as its argument.

Knudson [67] showed that, in the Arctic ...

`Knudson^{\cite{kn:gnus}}` showed ...

You can cite multiple sources with a single `\cite`, separating the keys by commas. The `\cite` command has an optional argument that adds a note to the citation.

Although they had disappeared from Fiji [4,15,36],  
Knudson [67, pages 222–333] showed that ...

... Fiji<sup>^{\cite{tom-ix,dick:ens,harry+d}}</sup>,  
`Knudson^{\cite[pages 222–333]{kn:gnus}}` ...

A citation key can be any sequence of letters, digits, and punctuation characters, except that it may not contain a comma. As usual in L<sup>A</sup>T<sub>E</sub>X, upper- and lowercase letters are considered to be different.

In the preceding examples, L<sup>A</sup>T<sub>E</sub>X has to determine that citation key `kn:gnus` corresponds to source label 67. How L<sup>A</sup>T<sub>E</sub>X does this depends on how you produce the list of sources. The best way to produce the source list is with a separate program called BIBT<sub>E</sub>X, described in Section 4.3.1. You can also produce it yourself, as explained in Section 4.3.2.

### 4.3.1 Using BIBTEX

BIBTEX is a separate program that produces the source list for a document, obtaining the information from a bibliographic database. To use BIBTEX, you must include in your LATEX input file a \bibliography command whose argument specifies one or more files that contain the database. The names of the database files must have the extension `bib`. For example, the command

```
\bibliography{insect,animal}
```

specifies that the source list is to be obtained from entries in the files `insect.bib` and `animal.bib`. There must be no space following the comma. Appendix B explains how to make bibliographic database files.

BIBTEX creates a source list containing entries for all the citation keys specified by \cite commands. The data for the source list is obtained from the bibliographic database, which must have an entry for every citation key. A \nocite command in the LATEX input file causes the specified entries to appear in the source list, but produces no output. For example, \nocite{g:nu,g:nat} causes BIBTEX to put bibliography database entries having keys `g:nu` and `g:nat` in the source list. The command \nocite{\*} causes all entries in the bibliographic database to be put in the source list. A \nocite command can go anywhere after the \begin{document} command, but it is fragile.

To use BIBTEX, your LATEX input file must contain a \bibliographystyle command. This command specifies the *bibliography style*, which determines the format of the source list. For example, the command

```
\bibliographystyle{plain}
```

specifies that entries should be formatted as specified by the `plain` bibliography style. The \bibliographystyle command can go anywhere after the \begin{document} command. LATEX's standard bibliography styles are:

`plain` Formatted more or less as suggested by van Leunen in *A Handbook for Scholars* [7]. Entries are sorted alphabetically and are labeled with numbers.

`unsrt` The same as `plain` except that entries appear in the order of their first citation.

`alpha` The same as `plain` except that source labels like "Knu66", formed from the author's name and the year of publication, are used.

`abbrv` The same as `plain` except that entries are more compact because first names, month names, and journal names are abbreviated.

Dozens of other bibliography styles exist, including ones that produce source lists in the formats used by a number of scientific journals. Consult the LATEX

*Companion* and the *Local Guide* to find out what styles are available. Documentation for the `BIBTeX` program explains how to create your own bibliography style.

The source list is normally formatted in what van Leunen calls a *compressed* style. The `openbib` document-class option causes it to be formatted in an *open* style. (Document-class options are specified by the `\documentclass` command; see Section 2.2.2.)

Once you've created an input file containing the appropriate `LATEX` commands, you perform the following sequence of steps to produce the final output:

- Run `LATEX` on the input file, which I assume is called `myfile.tex`. `LATEX` will complain that all your citations are undefined, since there is no source list yet.
- Run `BIBTeX` by typing something like `bibtex myfile`. (Consult your *Local Guide* to find out what you actually type.) `BIBTeX` will generate the file `myfile.bbl` containing `LATEX` commands to produce the source list.
- Run `LATEX` again on `myfile.tex`. `LATEX`'s output will now contain the source list. However, `LATEX` will still complain that your citations are undefined, since the output produced by a `\cite` command is based on information obtained from the source list the last time `LATEX` was run on the file.
- Run `LATEX` one more time on `myfile.tex`.

If you add or remove a citation, you will have to go through this whole procedure again to get the citation labels and source list right. But they don't have to be right while you're writing, so you needn't do this very often.

`BIBTeX` almost always produces a perfectly fine source list. However, it is only a computer program, so you may occasionally encounter a source that it does not handle properly. When this happens, you can usually correct the problem by modifying the bibliographic database—perhaps creating a special database entry just for this document. As a last resort, you can edit the `bbl` file that `BIBTeX` generated. (Of course, you should do this only when you are producing the final output.)

### 4.3.2 Doing It Yourself

A source list is created with the `thebibliography` environment, which is like the `enumerate` environment described in Section 2.2.4 except that:

- Each list item is begun with a `\bibitem` command whose argument is the citation key. (The `\bibitem` and `\cite` commands work much like the `\label` and `\ref` commands of Section 4.2.)

- The `thebibliography` environment has an argument that should be a piece of text the same width as or slightly wider than the widest item label in the source list.

Knudson [67] showed that, in the Arctic ...

⋮

## References

⋮

[67] D. E. Knudson. *1966 World Gnus Almanac*.  
Permafrost Press, Novosibirsk.

Knudson~\cite{kn:gnus} showed ...

⋮

\begin{thebibliography}{99}

⋮

\bibitem{kn:gnus} D. E. Knudson.

\emph{1966 World Gnus Almanac.}

⋮

\end{thebibliography}

In most type styles, “99” is at least as wide as all other two-digit numbers.

Instead of using numbers, you can choose your own labels for the sources by giving an optional argument to the `\bibitem` command.

Knudson [Knud 66] showed that, in the Arctic ...

⋮

## References

⋮

[Knud 66] D. E. Knudson. *1966 World Gnus Almanac*. Permafrost Press, Novosibirsk.

Knudson~\cite{kn:gnus} showed ...

⋮

\begin{thebibliography}{Dillo 83}

⋮

\bibitem[Knud 66]{kn:gnus} D. E. Knudson.

\emph{1966 World Gnus Almanac.}

⋮

\end{thebibliography}

In this example, “[Dillo 83]” should be the longest label. The optional argument of `\bibitem` is a moving argument.

As in any kind of cross-reference, citations are based upon the information gathered the previous time `LATEX` was run on the file. So, when you change the source list, you have to run `LATEX` two more times to change the citations.

## 4.4 Splitting Your Input

A large document requires a lot of input. Rather than putting the whole input in a single large file, you may find it more convenient to split the input into several smaller files. Regardless of how many separate files you use, there is one that is the *root* file; it is the one whose name you type when you run `LATEX`.

The `\input` command provides the simplest way to split your input into several files. The command `\input{gnu}` in the root file causes `LATEX` to insert the contents of the file `gnu.tex` right at the current spot in your manuscript—just as if the `\input{gnu}` command were removed from the root file and replaced by the contents of the file `gnu.tex`. (However, the input files are not changed.) The file `gnu.tex` may also contain an `\input` command, calling another file that may have its own `\input` commands, and so on.

Besides splitting your input into conveniently sized chunks, the `\input` command also makes it easy to use the same input in different documents. While text is seldom recycled in this way, you might want to reuse declarations. You can keep a file containing declarations that are used in all your documents, such as the definitions of commands and environments for your own logical structures (Section 3.4). You can even begin your root file with an `\input` command and put the `\documentclass` command in your declarations file.

Another reason for splitting the input into separate files is to run  $\text{\LaTeX}$  on only part of the document so, when you make changes, only the parts that have changed need to be processed. For this, you must use the `\include` command instead of `\input`. The two commands are similar in that `\include{gnu}` also specifies that the contents of the file `gnu.tex` should be inserted in its place. However, with the `\include` command, you can tell  $\text{\LaTeX}$  either to insert the file or to omit it and process all succeeding text as if the file had been inserted, numbering pages, sections, equations, etc., as if the omitted file's text had been included.

To run  $\text{\LaTeX}$  on only part of the document, the preamble must contain an `\includeonly` command whose argument is a list of files (first names only). The file specified by an `\include` command is processed only if it appears in the argument of the `\includeonly` command. Thus, if the preamble contains the command

```
\includeonly{gnu,gnat,gnash}
```

then an `\include{gnat}` command causes the file `gnat.tex` to be included, while the command `\include{rmadlo}` causes  $\text{\LaTeX}$  *not* to include the file `rmadlo.tex`, but to process the text following it as if the file had been included. More precisely, it causes  $\text{\LaTeX}$  to process the succeeding text under the assumption that the omitted file is exactly the same as it was the last time it was included.  $\text{\LaTeX}$  does not read an omitted file and is unaware of any changes made to the file since it was last included.

The entire root file is always processed. If the preamble does not contain an `\includeonly` command, then every `\include` command inserts its file. The command `\includeonly{}` (with a null argument) instructs  $\text{\LaTeX}$  to omit all `\include`'d files. An `\include` can appear only after the `\begin{document}` command.

The `\include` command has one feature that limits its usefulness: the included text always starts a new page, as does the text immediately following the `\include` command. It therefore works right only if the `\include`'d text and the text following it should begin on a new page—for example, if it consists of one or more complete chapters. While writing a chapter, you might want to split it into smaller files with `\include` commands. When you've finished writing that chapter, you can combine those files into a single one.

Another difficulty with the `\include` mechanism is that changing the docu-

ment may require reprocessing some unchanged `\include`'d files in order to get the correct numbering of pages, sections, etc. When skipping an `\include`'d file, the numbering in the succeeding text is based upon the numbering in the file's text the last time it was processed. Suppose that the root file contains the commands

```
\include{gnu}
\chapter{Armadillo}
```

and an `\includeonly` in the preamble causes the `\include` command to omit file `gnu.tex`. If the text in `gnu.tex` ended in Chapter 5 on page 56 the last time it was processed, even if you've added seven more chapters and sixty pages of text before the `\include` command since then, the `\chapter` command will produce Chapter 6 starting on page 57. In general, to make sure everything is numbered correctly, you must reprocess an `\include`'d file if a change to the preceding text changes the numbering in the text produced by that file.

When writing a large document, you should probably make each chapter a separate `\include`'d file. (You may find it convenient to enter the `\includeonly` command from the terminal, using the `\typein` command described in Section 4.6.) Process each file separately as you write or revise it, and don't worry about numbers not matching properly. If the inconsistent numbering gets too confusing, generate a coherent version by processing all the files at once. (You can do this by removing the `\includeonly` command before running L<sup>A</sup>T<sub>E</sub>X.) When you've finished making changes, run L<sup>A</sup>T<sub>E</sub>X on the entire document. You can then produce the final output either all at once by running L<sup>A</sup>T<sub>E</sub>X again on the entire document, or a few chapters at a time by using the `\includeonly` command.

## 4.5 Making an Index or Glossary

There are two steps in making an index or glossary: gathering the information that goes in it, and generating the L<sup>A</sup>T<sub>E</sub>X input to produce it. Section 4.5.1 describes the first step. The easiest way to perform the second step is with the *MakeIndex* program, described in Appendix A. Section 4.5.2 describes how to produce an index or glossary if you don't use *MakeIndex*.

### 4.5.1 Compiling the Entries

Compiling an index or a glossary is not easy, but L<sup>A</sup>T<sub>E</sub>X can help by writing the necessary information onto a special file. If the root file is named `myfile.tex`, index information is written on the “idx file” `myfile.idx`. L<sup>A</sup>T<sub>E</sub>X makes an `idx` file if the preamble contains a `\makeindex` command. The information on the file is written by `\index` commands; the command `\index{gnu}` appearing with the text for page 42 causes L<sup>A</sup>T<sub>E</sub>X to write

---

```
\indexentry{gnu}{42}
```

on the `idx` file. If there is no `\makeindex` command, the `\index` command does nothing. The `showidx` package causes L<sup>A</sup>T<sub>E</sub>X to print the arguments of all index commands in the margin.

The `\index` command produces no text, so you type

```
A gnat\index{gnat} with gnarled wings gnashed ...
```

to index this instance of “gnat”. It’s best to put the `\index` command next to the word it references, with no space between them; this keeps the page number from being off by one if the word ends or begins a page. I find it best to put index entries on a separate line, as in

```
When in the Course of
 \index{human events}%
 \index{events, human}%
human events, it becomes necessary for one people
```

This use of the % character is explained on page 19.

The procedure for making a glossary is completely analogous. In place of `\index` there is a `\glossary` command. The `\makeglossary` command produces a file with the extension `glo` that is similar to the `idx` file except with `\glossaryentry` entries instead of `\indexentry` entries.

The argument of `\index` or `\glossary` can contain any characters, including special characters like `\` and `$`. However, curly braces must be properly balanced, each `{` having a matching `}`. The `\index` and `\glossary` commands are fragile. Moreover, an `\index` or `\glossary` command should not appear in the argument of any other command if its own argument contains any of L<sup>A</sup>T<sub>E</sub>X’s ten special characters (Section 2.1).

### 4.5.2 Producing an Index or Glossary by Yourself

If you don’t use the *MakeIndex* program, you can use the `theindex` environment to produce an index in two-column format. Each main index entry is begun by an `\item` command. A subentry is begun with `\subitem`, and a subsubentry is begun with `\subsubitem`. Blank lines between entries are ignored. An extra vertical space is produced by the `\indexspace` command, which is usually put before the first entry that starts with a new letter.

```
gnats 13, 97
gnus 24, 37, 233
 bad, 39, 236
 very, 235
 good, 38, 234
.
harmadillo 99, 144

```

|                                                                                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\item gnats 13, 97</code><br><code>\item gnus 24, 37, 233</code><br><code>  \subitem bad, 39, 236</code><br><code>    \subsubitem very, 235</code><br><code>  \subitem good, 38, 234</code><br><code>\indexspace</code><br><code>\item harmadillo 99, 144</code> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

There is no environment expressly for glossaries. However, the `description` environment of Section 2.2.4 may be useful.

## 4.6 Keyboard Input and Screen Output

When creating a large document, it's often helpful to leave a reminder to yourself in the input file—for example, to note a paragraph that needs rewriting. The use of the `%` character for putting comments into the text is described in Section 2.2.1. However, a passive comment is easy to overlook, so `LATEX` provides the `\typeout` command for printing messages on the terminal. In the examples in this section, the left column shows what is produced on the terminal by the input in the right column; the oval represents the terminal.

Don't forget to revise this!

`\typeout{Don't forget to revise this!}`

Remember that everything `LATEX` writes on the terminal is also put in the `log` file.

It is sometimes useful to type input to `LATEX` directly from your keyboard—for example, to enter an `\includeonly` command. This is done with a `\typein` command, such as the following:

Enter 'includeonly', boss!

`\typein {Enter 'includeonly', boss!}`

`\@typein=`

When this appears on your terminal, `LATEX` is waiting for you to enter a line of input, ended by pressing the *return* key. `LATEX` then processes what you typed just as if it had appeared in the input file in place of the `\typein` command.

The `\typein` command has an optional first argument, which must be a command name. When this optional argument is given, instead of processing your typed input at that point, `LATEX` defines the specified command to be equivalent to the text that you have typed.

Enter lover's name.

`\typein [\lover]{Enter lover's name.}`

`\lover=`

I love \lover\ very much.

Typing `Chris` and pressing the return key causes the `\typein` command to define the command `\lover` to be equivalent to `Chris`—just like the `\newcommand` or `\renewcommand` commands of Section 3.4. Thus, the input following this `\typein` command would produce

I love Chris very much.

The argument of the `\typeout` or `\typein` command is a moving argument. Both of these commands are fragile.

## 4.7 Sending Your Document

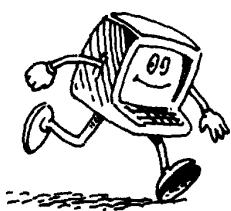
Today, you are likely to send a document electronically—by e-mail or on a diskette—rather than on paper. Putting a file in an e-mail message or on a diskette is no problem. However, to process your document, L<sup>A</sup>T<sub>E</sub>X must read a number of files in addition to the main input file. Some files, such as the `toc` file (Section 4.1), are generated by L<sup>A</sup>T<sub>E</sub>X itself. The rest must be sent if the recipient does not already have them. These files include the `bb1` file (Section 4.3.1), files required by any nonstandard packages the document uses, and files that are read by `\input` or `\include` commands. You can combine all these files into a single L<sup>A</sup>T<sub>E</sub>X input file by using the `filecontents` environment. Instead of sending the file `name.tex` separately, put the environment

```
\begin{filecontents}{name.tex}
contents of file
\end{filecontents}
```

at the beginning of your input file. When L<sup>A</sup>T<sub>E</sub>X is run on this input file, it writes the file `name.tex` if no file by that name already exists. If such a file does exist, L<sup>A</sup>T<sub>E</sub>X prints a warning message. You can put any number of `filecontents` environments in an input file, but they must all precede the `\documentclass` command.

To find out what files you need to send, put a `\listfiles` command in the preamble. L<sup>A</sup>T<sub>E</sub>X will then print out a list of the files it reads when processing the document. The list will not include files generated by L<sup>A</sup>T<sub>E</sub>X, and it will identify standard files that are always present on a computer that runs L<sup>A</sup>T<sub>E</sub>X. However, the list may not tell you in what directories the files are. The terminal output might include complete file names. (As explained in Section 8.1, when L<sup>A</sup>T<sub>E</sub>X starts to read a file, it writes the file's name on the terminal and in the log file.) You can also consult the *Local Guide* to find out what directories contain the files read by packages.

When sending files by e-mail, don't forget about the problem of lines beginning with `From:` that is discussed in Section 2.4.



## CHAPTER 5

# Other Document Classes



## 5.1 Books

Books differ from reports mainly in their front and back matter. The front matter of a book usually includes a half-title page, a main title page, a copyright page, a preface or foreword, and a table of contents. It may also contain acknowledgements, a dedication, a list of figures, a list of other books in the same series, and so on. The back matter usually includes an index and may contain an afterword, acknowledgements, a bibliography, a colophon, etc.

The book document class does not attempt to provide commands for all the logical structures that could appear in a book's front and back matter. It is just the same as the `report` class except for the differences described below. Individual publishers might have packages with additional commands for the particular structures they want in their books.

The front matter, main matter (the main body of the book, including appendices), and back matter are begun with the three commands `\frontmatter`, `\mainmatter`, and `\backmatter`, respectively. In the standard `book` class, front matter pages are numbered with roman numerals; main and back matter pages are numbered with arabic numerals. In the front and back matter, the `\chapter` command does not produce a chapter number, but it does make a table of contents entry. Thus, you can begin a preface with

```
\chapter{Preface}
```

Only the `*` forms of other sectioning commands should be used in the front and back matter (see Section C.4.1).

Two-sided printing is the norm for books, so the `twoside` option is the default in the `book` class. Another default option is `openright`, which causes new chapters to begin on a right-hand page. The contrary option, `openany`, is the default for the `report` class.

A book can have more than one title page, and each title page can contain a variety of information besides the title and author—for example, the publisher, series, and illustrator. The `\maketitle` command is therefore of little use in a book.

You will probably have to do a considerable amount of customizing of the style parameters for a book—especially of the page-style parameters, described in Section C.5.3. The time this takes should be infinitesimal compared with the time you spend writing the book.

## 5.2 Slides

The `slides` document class is used for making slides. Slides are usually printed or photocopied onto transparencies for projection on a screen.

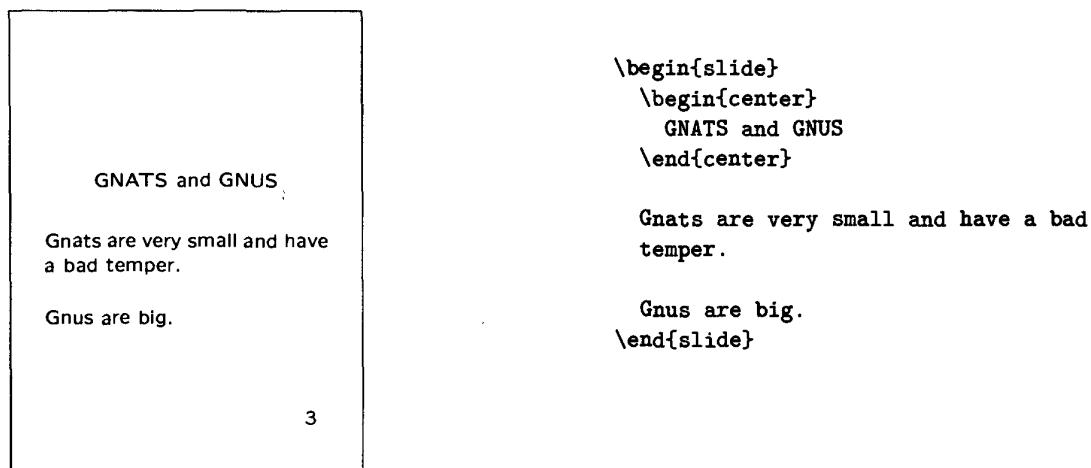
Producing good slides requires visual formatting, which means that L<sup>A</sup>T<sub>E</sub>X is not well suited for the task. Here are a few reasons why you may want to use L<sup>A</sup>T<sub>E</sub>X anyway:

- Your slides are based on material from a L<sup>A</sup>T<sub>E</sub>X document.
- Your slides have a lot of mathematical formulas.
- You don't make slides often enough to bother learning how to use another system.

The `slides` document class's default fonts were specially designed to be easy to read from a distance. To my knowledge, no other system makes slides that are so readable. The `slides` class's fonts are much larger than the usual ones; `\normalsize` produces roughly the same size characters as `\LARGE` in other document classes (Section 6.7.1). Moreover, the roman family is similar to the ordinary sans serif family. The only type families generally available are roman and typewriter, and the only type shapes are upright and italic. (See Section 3.1.) The `\emph` command works as usual.

### 5.2.1 Slides and Overlays

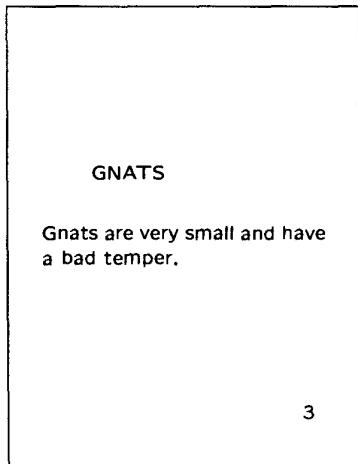
An individual slide is produced with a `slide` environment:



```
\begin{slide}
 \begin{center}
 GNATS and GNUS
 Gnats are very small and have a bad temper.
 Gnus are big.
 \end{center}
\end{slide}
```

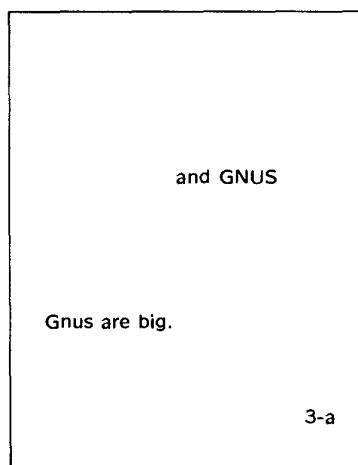
The text appearing on a slide is produced with ordinary L<sup>A</sup>T<sub>E</sub>X commands. Any commands that make sense for slides can be used. Commands that *don't* make sense include sectioning commands, `figure` and `table` environments, and page-breaking commands. The latter make no sense in a slide because each slide must fit on a single page. To make colored slides, use the `color` package described in Section 7.3.

The `overlay` environment is used for an overlay—a slide that is meant to be put on top of another slide. It is the same as the `slide` environment except for the numbering; the first overlay following slide number 3 is numbered “3-a”, the second one is numbered “3-b”, and so on. The best way to get the text on the slide and the overlay to line up properly is to have all the text on both of them and make text that isn’t supposed to appear invisible by coloring it white (Section 7.3).



```
\begin{slide}
\begin{center}
GNATS \textcolor{white}{and GNUS}
\end{center}
Gnats are very small
and have a bad temper.

\textcolor{white}{Gnus are big.}
\end{slide}
```



```
\begin{overlay}
\begin{center}
\textcolor{white}{GNATS} and GNUS
\end{center}
\textcolor{white}{Gnats are very small
and have a bad temper.}

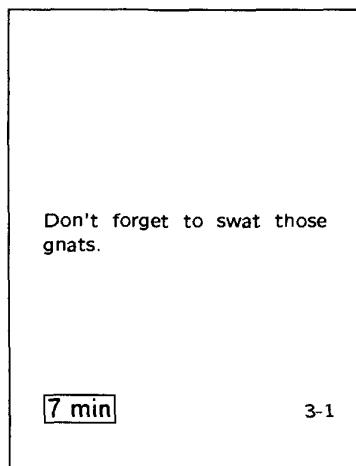
Gnus are big.
\end{overlay}
```

Superimposing this slide and overlay yields the preceding slide (plus an extra “-a” next to the slide number). If the `color` package is not available on your computer, you can use the commands of Section 6.4.2 to produce the blank spaces.

### 5.2.2 Notes

When giving a lecture, it helps if you put notes to yourself in with the slides to remind you of what to say. The `note` environment produces a one-page note. Notes that follow slide number 3 are numbered “3-1”, “3-2”, etc. I always make a note to accompany each slide. (You can print the slides and notes separately by using the `\onlyslides` and `\onlynotes` commands described below.)

A good lecturer plans a talk carefully and does not run out of time. The `slides` document class’s `clock` option helps you keep track of time during your lecture. Right before or after each slide, put an `\addtime` command giving the number of seconds you should spend on that slide. The total amount of time (in minutes) that you should have taken so far will be printed at the bottom of each note.



```
\documentclass[clock]{slides}
...
\addtime{180} % Slide 1: 3 minutes
...
\addtime{150} % Slide 2: 2-1/2 minutes
...
\addtime{120} % Slide 3: 2 minutes

\begin{note}
Don't forget to swat those gnats.
\end{note}
```

`LATEX` will also type out on your terminal the total elapsed time at the end of the document.

You can reset the elapsed time with the `\setttime` command. The command `\setttime{120}` sets the total elapsed time to 120 seconds (2 minutes). Do not put an `\addtime` or `\resetttime` command inside a `slide`, `overlay`, or `note` environment.

### 5.2.3 Printing Only Some Slides and Notes

For making corrections, it’s convenient to generate only some of the slides and notes from your input file. The command

```
\onlyslides{4,7-13,23}
```

in the preamble will cause `LATEX` to generate only slides numbered 4, 7–13 (inclusive), and 23, plus all of their overlays. The slide numbers in the argument

must be in ascending order, and can include nonexistent slides—for example, you can type

```
\onlyslides{10-9999}
```

to produce all but the first nine slides. The argument of the `\onlyslides` command must be nonempty.

There is also an analogous `\onlynotes` command to generate a subset of the notes. Notes numbered 11-1, 11-2, etc. will all be generated by specifying page 11 in the argument of the `\onlynotes` command. If the input file has an `\onlyslides` command but no `\onlynotes` command, then no notes are produced, and vice versa. Including both an `\onlyslides` and an `\onlynotes` command has the expected effect of producing only the specified slides and notes.

#### 5.2.4 Other Text

Your input file can include text that appears outside a `slide`, `overlay`, or `note` environment. Such other text is formatted like a slide, but without a slide number. It can be used to make a title page. This other text is always printed, even when an `\onlyslides` or `\onlynotes` command is given.

Text outside a slide or note differs from text inside one only in that page-breaking commands are allowed. Use a `\pagebreak` command if you want to tell L<sup>A</sup>T<sub>E</sub>X where to start a new page in the middle of the text.

### 5.3 Letters

For making letters—the kind that are put in an envelope and mailed—L<sup>A</sup>T<sub>E</sub>X provides the `letter` document class. To use this class, begin your input file with a `\documentclass` command having `letter` as its main (nonoptional) argument (Section 2.2.2).

You can make any number of letters with a single input file. Your name and address, which are likely to be the same for all letters, are specified by declarations. The return address is declared by an `\address` command, with multiple output lines separated by `\` commands.

```
\address{1234 Ave. \ of the Armadillos\\
Gnu York, G.Y. 56789}
```

The `\signature` command declares your name, as it appears at the end of the letter, with the usual `\` commands separating multiple lines.

```
\signature{R. (Ma) Dillo \\ Director of Cuisine}
```

These declarations are usually put in the preamble, but they are ordinary declarations with the customary scoping rules and can appear anywhere in the document.

Each letter is produced by a separate `letter` environment, having the name and address of the recipient as its argument. The argument of the `letter` environment is a moving argument. The letter itself begins with an `\opening` command that generates the salutation.

1234 Ave. of the Armadillos  
Gnu York, G.Y. 56789

July 4, 1996

```
\begin{letter}{Dr. \ G. Nathaniel Picking \\
Acme Exterminators \\ 33 Swat Street \\
Hometown, Illinois 62301}
```

Dr. G. Nathaniel Picking  
Acme Exterminators  
33 Swat Street  
Hometown, Illinois 62301

```
\opening{Dear Nat,}
```

I'm afraid that the armadillo problem  
is still with us. I did everything

...

Dear Nat,

I'm afraid that the armadillo problem is still with  
us. I did everything ...

The return address is determined by the `\address` declaration; L<sup>A</sup>T<sub>E</sub>X supplies the date. An `\address` and/or `\signature` command that applies just to this letter can be put between the `\begin{letter}` and the `\opening` command.

The main body of the letter is ordinary L<sup>A</sup>T<sub>E</sub>X input, but commands like `\section` that make no sense in a letter should not be used. The letter closes with a `\closing` command.

... and I hope you can get rid of the nasty beasts  
this time.

... and I hope you can get rid of the nasty  
beasts this time.

Best regards,

```
\closing{Best regards,}
```

R. (Ma) Dillo  
Director of Cuisine

The name comes from the `\signature` declaration.

The `\cc` command can be used after the closing to list the names of people to whom you are sending copies.

cc: Bill Clinton  
George Bush

```
\cc {Bill Clinton \\ George Bush}
```

There's a similar `\encl` command for a list of enclosures.

Additional text after the closing must be preceded by a `\ps` command. This command generates no text—you'll have to type the “P.S.” yourself—but is needed to format the additional text correctly.

You can change the date on your letters by using `\renewcommand` (Section 3.4.1) to redefine `\today`. Put the `\renewcommand` in the preamble to change the date on all letters; put it right before the `\opening` command to change the date on a single letter.

|                                                     |                                                                             |
|-----------------------------------------------------|-----------------------------------------------------------------------------|
| 1234 Ave. of the Armadillos<br>Gnu York, G.Y. 56789 | <code>\begin{letter}{Mr.\ A. Oop \\ Kingdom of Moo}</code>                  |
| Thursday, 15842 B.C.                                | <code>\renewcommand{\today}{Thursday, 15842 B.C.}</code>                    |
| Mr. A. Oop<br>Kingdom of Moo                        | <code>\opening{Dear Mr.\ Oop:}</code>                                       |
| Dear Mr. Oop:                                       | <code>In your last shipment of stegosaur steaks,</code><br><code>...</code> |
| In your last shipment of stegosaur steaks, ...      |                                                                             |

Redefining `\today` will not alter the postmark printed on the envelope by the post office.

A `\makelabels` command in the preamble will cause L<sup>A</sup>T<sub>E</sub>X to print a list of mailing labels, one for each `letter` environment, in a format suitable for xerographic copying onto “peel-off” labels. A mailing label without a corresponding letter is produced by an empty `letter` environment—one with nothing between the argument and the `\end{letter}` command.

There may be other features for making letters—especially if you are using L<sup>A</sup>T<sub>E</sub>X at a company or university. For example, you may be able to format letters for copying onto the company letterhead. Consult the *Local Guide* for more information.

## CHAPTER 6

# Designing It Yourself



The preceding chapters describe L<sup>A</sup>T<sub>E</sub>X commands and environments for specifying a document's logical structure. This chapter explains how to specify its visual appearance. Before reading it, you should review the discussion in Section 1.4 of the dangers of visual design. Commands specifying the visual appearance of the document are usually confined to the preamble, either as style declarations or in the definitions of commands and environments for specifying logical structures. The notable exceptions are the line- and page-breaking commands of Section 6.2 and the picture-drawing commands of Section 7.1.

## 6.1 Document and Page Styles

### 6.1.1 Document-Class Options

The style of the document can be changed by using the optional argument of the `\documentclass` command to specify style options. The `11pt`, `12pt`, `twoside`, and `twocolumn` options are described in Section 2.2.2. Three other standard options are:

`titlepage` Causes the `\maketitle` command to generate a separate title page and the `abstract` environment to make a separate page for the abstract. This option is the default for the `report` document class.

`leqno` Causes the formula numbers produced by the `equation` and `eqnarray` environments to appear on the left instead of the right.

`fleqn` Causes formulas to be aligned on the left, a fixed distance from the left margin, instead of being centered.

A complete list of the standard options appears in Section C.5.1. The *Local Guide* will tell you if there are any other options available on your computer.

The `twocolumn` option is used for a document that consists mostly of double-column pages. It sets various style parameters, such as the amount of paragraph indentation, to values appropriate for such pages. You may want double-column pages for just part of the document, such as a glossary. The `\twocolumn` declaration starts a new page and begins producing two-column output. It does not change any style parameters except the ones required to produce two-column output. The inverse `\onecolumn` declaration starts a new page and begins producing single-column output.

In books, it is conventional for the height of the text to be the same on all full pages. The `\flushbottom` declaration makes all text pages the same height, adding extra vertical space when necessary to fill out the page. The `\raggedbottom` declaration has the opposite effect, letting the height of the text vary a bit from page to page. The default is `\raggedbottom` except when the `twoside` option is in effect. You can change the default by putting the appropriate declaration in the preamble.

### 6.1.2 Page Styles

A page of output consists of three units: the *head*, the *body*, and the *foot*. In most pages of this book, the head contains a page number, a chapter or section title, and a horizontal line, while the foot is empty; but in the table of contents and the preface, the page head is empty and the foot contains the page number. The body consists of everything between the head and foot: the main text, footnotes, figures, and tables.

The information in the head and foot, which usually includes a page number, helps the reader find his way around the document. You can specify arabic page numbers with a `\pagenumbering{arabic}` command and roman numerals with a `\pagenumbering{roman}` command, the default being arabic numbers. The `\pagenumbering` declaration resets the page number to one, starting with the current page. To begin a document with pages i, ii, etc. and have the first chapter start with page 1, put `\pagenumbering{roman}` anywhere before the beginning of the text and `\pagenumbering{arabic}` right after the first `\chapter` command.

Page headings may contain additional information to help the reader. They are most useful in two-sided printing, since headings on the two facing pages convey more information than the single heading visible with one-sided printing. Page headings are generally not useful in a short document, where they tend to be distracting rather than helpful.

The *page style* determines what goes into the head and foot; it is specified with a `\pagestyle` declaration having the page style's name as its argument. There are four standard page styles:

**plain** The page number is in the foot and the head is empty. It is the default page style for the `article` and `report` document classes.

**empty** The head and foot are both empty. L<sup>A</sup>T<sub>E</sub>X still assigns each page a number, but the number is not printed.

**headings** The page number and other information, determined by the document style, is put in the head; the foot is empty.

**myheadings** Similar to the `headings` page style, except you specify the “other information” that goes in the head, using the `\markboth` and `\markright` commands described below.

The `\pagestyle` declaration obeys the normal scoping rules. What goes into a page's head and foot is determined by the page style in effect at the end of the page, so the `\pagestyle` command usually comes after a command like `\chapter` that begins a new page.

The `\thispagestyle` command is like `\pagestyle` except it affects only the current page. Some commands, such as `\chapter`, change the style of the current

page. These changes can be overridden with a subsequent `\thispagestyle` command.

The contents of the page headings in the `headings` and `myheadings` styles are set by the following commands:

```
\markboth{left_head}{right_head}
\markright{right_head}
```

The `left_head` and `right_head` arguments specify the information to go in the page heads of left-hand and right-hand pages, respectively. In two-sided printing, specified with the `twoside` document-class option, the even-numbered pages are the left-hand ones and the odd-numbered pages are the right-hand ones. In one-sided printing, all pages are considered to be right-hand ones.

In the `headings` page style, the sectioning commands choose the headings for you; Section C.5.3 explains how to use `\markboth` and `\markright` to override their choices. In the `myheadings` option, you must use these commands to set the headings yourself. The arguments of `\markboth` and `\markright` are processed in LR mode; they are moving arguments.

### 6.1.3 The Title Page and Abstract

The `\maketitle` command produces a title page when the `titlepage` document-class option is in effect; it is described in Sections 2.2.2 and C.5.4. You can also create your own title page with the `titlepage` environment. This environment creates a page with the `empty` page style, so it has no printed page number or heading. It causes the following page to be numbered 1.

You are completely responsible for what appears on a title page made with the `titlepage` environment. The following commands and environments are useful in formatting a title page: the type-size-changing commands of Section 6.7.1, the type-style-changing commands of Section 3.1, and the `center` environment of Section 6.5. Recall also that the `\today` command, described in Section 2.2.1, generates the date. You will probably produce several versions of your document, so it's important to include a date on the title page.

An abstract is made with the `abstract` environment.

#### Abstract

The mating habits of insects are quite different from those of large mammals.

```
\begin{abstract}
```

```
The mating habits of insects are quite
different from those of large mammals.
\end{abstract}
```

The abstract is placed on a separate page if the `titlepage` document-class option is in effect; otherwise, it acts like an ordinary displayed-paragraph environment.

### 6.1.4 Customizing the Style

If you don't like the style of the output produced by L<sup>A</sup>T<sub>E</sub>X with the standard style options, you should check the *L<sup>A</sup>T<sub>E</sub>X Companion* and the *Local Guide* to see if there are other options available. If there aren't, you must change the style of your document yourself. Changing the style means changing the way the standard structures such as paragraphs and itemized lists are formatted, not creating new structures. Section 3.4 describes how to define new logical structures.

Before changing your document's style, remember that many authors make elementary errors when they try to design their own documents. The only way to avoid these errors is by consulting a trained typographic designer or reading about typographic design. All I can do here is warn you against the very common mistake of making lines that are too wide to read easily—a mistake you won't make if you follow this suggestion: *Use lines that contain no more than 75 characters, including punctuation and spaces.*

The style of a particular document can be customized by adding declarations to its preamble. If the same style modifications are used in several documents, it is more convenient to put them in a separate package. A package is created by writing the appropriate declarations in a **sty** file—a file whose first name is the package name and whose extension is **sty**. For example, to define a package named *vacation*, you would create the file *vacation.sty*. The command `\usepackage{vacation}` would then cause L<sup>A</sup>T<sub>E</sub>X to read the file *vacation.sty*.

When reading a package's **sty** file, T<sub>E</sub>X regards an **C** character as a letter, so it can be part of a command name like `\Qlisti`. Such a command name cannot be used in your document, since T<sub>E</sub>X would interpret it as the command `\Q` followed by the text characters *listi*. Many of L<sup>A</sup>T<sub>E</sub>X's internal commands have an **C** in their name to prevent their accidental use within the document; these include some parameters described in Appendix C that you may want to change.

The simplest way to modify a document's style is by changing parameters such as the ones that control the height and width of the text on the page. L<sup>A</sup>T<sub>E</sub>X's style parameters are described in this chapter and in Appendix C. Other modifications require redefining L<sup>A</sup>T<sub>E</sub>X commands. For example, if you want to change the style of chapter headings, you will have to redefine the `\chapter` command. If the change is complicated—for example, if you want the chapter heading to list the titles of all preceding chapters—then you will need to learn advanced T<sub>E</sub>X commands. However, it isn't hard to figure out how to make most of the changes you are likely to want. To illustrate the process, I will now describe how you would make a typical style change: printing chapter titles in a bold sans serif type style, rather than in the standard bold roman style.

You have to change the definition of `\chapter`, so the first problem is finding that definition. Commands that exist in all document classes are usually *preloaded*. The file **source2e.tex** lists the names of the files in which the defi-

nitions of preloaded commands can be found. (Consult the *Local Guide* to find out in what directory these files are.) Commands like `\chapter` that exist only in some document classes or are defined by a package are not preloaded. The definitions of these commands are in files read by L<sup>A</sup>T<sub>E</sub>X when it processes your document. Section 4.7 explains how to find out what files L<sup>A</sup>T<sub>E</sub>X reads.

For this example, I assume that your document uses the `report` document class. You will find that L<sup>A</sup>T<sub>E</sub>X reads the file `report.cls`. Searching this file for “`\chapter`”, you will come across

```
\newcommand{\chapter}{\if@openright\cleardoublepage\else...
 \thispagestyle{plain}%
 \global\@topnum\z@
 \global\@afterindentfalse
 \secdef\@chapter\@schapter}
```

This is where the `\chapter` command is defined. (Macho T<sub>E</sub>X programmers sometimes remove the braces around the first argument of `\newcommand`; don’t do it yourself.) Looking at this definition, it’s not hard to guess that the chapter heading is produced by `\@chapter` or `\@schapter`. Immediately below the definition of `\chapter` is

```
\def\@chapter[#1]{#2}{...}
```

This is where `\@chapter` is defined. (Macho T<sub>E</sub>X programmers also use the T<sub>E</sub>X command `\def` to define commands; don’t you do it.) Examining this definition will lead you to suspect that the heading is produced by the `\@makechapterhead` command. The definition of `\@makechapterhead` contains two `\bfseries` commands. An obvious guess is that adding `\sffamily` commands right before or after these `\bfseries` commands will produce bold sans serif chapter headings. So, you will create a file named `sfchap.sty` containing the modified definition of `\@makechapterhead`, add a `\usepackage{sfchap}` command to the preamble of your document, and run L<sup>A</sup>T<sub>E</sub>X. You will then discover that you have guessed right; L<sup>A</sup>T<sub>E</sub>X has produced the bold sans serif headings you wanted.

This detective work was good practice, but there’s a way to avoid much of it. Comments at the beginning of `report.cls` indicate that this file was generated from a source file named `classes.dtx`. (The file `source2e.tex` also directs you to `classes.dtx`.) A `dtx` file contains comments and additional formatting commands. You can run a `dtx` file through L<sup>A</sup>T<sub>E</sub>X to produce a printed version, or you can just read the `dtx` file itself. The file `classes.dtx` reveals that the `\@schapter` command prints unnumbered chapter headings. (Such headings are produced by the `\chapter*` command described in Section C.4.1.) Aided by the comments, you should now be able to figure out how to change the way unnumbered chapter headings are printed.

This example shows how to modify a command; modifying an environment is similar. To find its definition, you need to know that some environments are

defined with  $\text{\TeX}$ 's `\def` command instead of `\newenvironment`. For example, the `equation` environment is defined by defining `\equation`, which is executed by the `\begin{equation}` command, and `\endequation`, which is executed by `\end{equation}`.

Remember that before creating your own package of style changes, you should check the *L<sup>A</sup>T<sub>E</sub>X Companion* and the *Local Guide* to see if someone has already created a document class or package that does what you want.

## 6.2 Line and Page Breaking

$\text{\TeX}$  usually does a good job of breaking text into lines and pages, but it sometimes needs help. Don't worry about line and page breaks until you prepare the final version. Most of the bad breaks that appear in early drafts will disappear as you change the text. Many L<sup>A</sup>T<sub>E</sub>X users waste time formatting when they should be writing. **Don't worry about line and page breaks until you prepare the absolutely final version.**

### 6.2.1 Line Breaking

Let's return to the line-breaking problem that we inserted into the `sample2e` input file in Section 2.3. Recall that it produced the following warning message:

```
Overfull \hbox (10.51814pt too wide) in paragraph at lines 195--200
[]\OT1/cmr/m/n/10 Mathematical for-mu-las may also be dis-played. A dis-played
for-mula is gnomonly
```

$\text{\TeX}$  could not find a good place to break the line and left the word “gnomonly” extending past the right margin. The first line of this warning message states that the output line extends 10.51814 points past the right margin—a point is about 1/72<sup>nd</sup> of an inch—and is in the paragraph generated by lines 195 through 200 of the input file. The next part of the message shows the input that produced the offending line, except  $\text{\TeX}$  has inserted a “–” character every place that it's willing to hyphenate a word. The `draft` document-class option causes  $\text{\TeX}$  to put a black box next to the extra-long line, making it easy to find.

$\text{\TeX}$  is quite good at hyphenating words; an English-language version never<sup>1</sup> incorrectly hyphenates an English word and usually finds all correct possibilities. However, it does miss some. For example, it does not know how to hyphenate the word *gnomonly* (which isn't a very gnomonly used word), nor can it hyphenate *gnomon*.

A `\-` command tells  $\text{\TeX}$  that it is allowed to hyphenate at that point. We could correct our sample hyphenation problem by changing `gnomonly` to `gn\-\omonly`, allowing  $\text{\TeX}$  to break the line after *gno*. However, it's better to change it to `gn\-\om\-\ly`, which also allows  $\text{\TeX}$  to break right before the *ly*.

---

<sup>1</sup>Well, hardly ever.

While  $\text{\TeX}$  will still break this particular sample line after *gno*, further changes to the text might make *gnomon-ly* better.

$\text{\TeX}$  will not hyphenate a word with a nonletter in the middle, where it treats any sequence of nonspace characters as a single word. While it hyphenates *ra-di-a-tion* properly, it does not hyphenate *x-radiation*—though it will break a line after the *x*. You must type *x-ra\-\di\-\a\-\tion* for  $\text{\TeX}$  to consider all possible hyphenation points. However, it is generally considered a bad idea to hyphenate a hyphenated compound; you should do so only when there is no better alternative.

When writing a paper about sundials, in which the word *gnomon* appears frequently, it would be a nuisance to type it as *gno\-\mon* everywhere it is used. You can teach  $\text{\TeX}$  how to hyphenate words by putting one or more *\hyphenation* commands in the preamble. The command

```
\hyphenation{gno-mon gno-mons gno-mon-ly}
```

tells  $\text{\TeX}$  how to hyphenate *gnomon*, *gnomons*, and *gnomonly*—but it still won't know how to hyphenate *gnomonic*.

While it's very good at hyphenating English, an English-language version of  $\text{\LaTeX}$  does not respect the hyphenation rules of other languages. For example, it uses the English hyphenation *re-spect* rather than the Romanian *res-pect*. If your document contains only a few phrases in Romanian, you can correct hyphenation errors as they occur by using *\hyphenation* or *\-* commands to tell  $\text{\TeX}$  where it can hyphenate a word. If your document contains a lot of text in Romanian or some other language(s), use the *babel* package described in the  *$\text{\LaTeX}$  Companion*.

Not all line-breaking problems can be solved by hyphenation. Sometimes there is just no good way to break a paragraph into lines.  $\text{\TeX}$  is normally very fussy about line breaking; it lets you solve the problem rather than producing a paragraph that doesn't meet its high standards. There are three things you can do when this happens. The first is to rewrite the paragraph. However, having carefully polished your prose, you may not want to change it just to produce perfect line breaks.

The second way to handle a line-breaking problem is to use a *sloppypar* environment or *\sloppy* declaration, which direct  $\text{\TeX}$  not to be so fussy about where it breaks lines. Most of the time, you just enclose the entire paragraph that contains the bad line break between *\begin{sloppypar}* and *\end{sloppypar}* commands. However, sometimes it's easier to use a *\sloppy* declaration. To explain how to use this declaration, it helps to introduce the concept of a *paragraph unit*. A paragraph unit is a portion of text that is treated as a single string of words to be broken into lines at any convenient point. For example, a paragraph containing a displayed equation would consist of two paragraph units—the parts of the paragraph that come before and after the equation. (Since the equation itself can't be broken across lines, it is not a paragraph unit.) Similarly, each

item in a list-making environment begins a new paragraph unit.

*T**EX* does its line breaking for a paragraph unit when it encounters the command or blank line that ends the unit, based upon the declarations in effect at that time. So, the scope of the `\sloppy` declaration should include the command or blank line that ends the paragraph unit with the bad line break. You can either delimit the scope of the `\sloppy` declaration with braces, or else use a countermanding `\fussy` declaration that restores *T**EX* to its ordinary compulsive self. The `\begin{sloppypar}` command is equivalent to a blank line followed by `\sloppy`, and `\end{sloppypar}` is equivalent to a blank line followed by a `}``.

The third way to fix a bad line break is with a `\linebreak` command, which forces *T**EX* to break the line at that spot. The `\linebreak` is usually inserted right before the word that doesn't fit. An optional argument converts the `\linebreak` command from a demand to a request. The argument must be a digit from 0 through 4, a higher number denoting a stronger request. The command `\linebreak[0]` allows *T**EX* to break the line there, but neither encourages nor discourages its doing so, while `\linebreak[4]` forces the line break just like an ordinary `\linebreak` command. The arguments 1, 2, and 3 provide intermediate degrees of insistence, and may succeed in coaxing *T**EX* to overcome a bad line break. They can also be used to help *T**EX* find the most aesthetically pleasing line breaks. The `\linebreak[0]` command allows a line break where it would normally be forbidden, such as within a word.

Both of these methods handle line-breaking problems by sweeping them under the rug. The “lump in the carpet” that they may leave is one or more lines with too much blank space between words. Such a line will produce an “Underfull `\hbox`” warning message.

Although unwanted line breaks are usually prevented with the `~` and `\mbox` commands described in Section 2.2.1, *L**A**T**E**X* also provides a `\nolinebreak` command that forbids *T**EX* from breaking the line at that point. Like the `\linebreak` command, `\nolinebreak` takes a digit from 0 through 4 as an optional argument to convert the prohibition into a suggestion that this isn't a good place for a line break—the higher the number, the stronger the suggestion. A `\nolinebreak[0]` command is equivalent to `\linebreak[0]`, and `\nolinebreak[4]` is equivalent to `\nolinebreak`.

A `\linebreak` command causes *T**EX* to justify the line, stretching the space between words so the line extends to the right margin. The `\newline` command ends a line without justifying it.

I can think of no good reason why you would want to make a short line like this in the middle of a paragraph, but perhaps you can think of one.

I can think of no good reason why you would want to make a short line like this  
`\newline`  
in the middle of a paragraph,  
but perhaps you can think of one.

You can type `\\"`, which is the usual L<sup>A</sup>T<sub>E</sub>X command for ending a line, in place of `\newline`. In fact, L<sup>A</sup>T<sub>E</sub>X provides the `\newline` command only to maintain a complete correspondence between the line-breaking commands and the page-breaking commands described below.

The `\linebreak`, `\nolinebreak`, and `\newline` commands can be used only in paragraph mode. They are fragile commands. See Section C.1.1 if a `[` follows a `\linebreak` or `\nolinebreak` command that has no optional argument.

Remember, **don't worry about line breaks until you prepare the ultimate, absolutely final version.**

### 6.2.2 Page Breaking

T<sub>E</sub>X is as fussy about page breaks as it is about line breaks. As with line breaking, sometimes T<sub>E</sub>X can find no good place to start a new page. A bad page break usually causes T<sub>E</sub>X to put too little rather than too much text onto the page. When the `\flushbottom` declaration (Section 6.1.1) is in effect, this produces a page with too much extra vertical space; with the `\raggedbottom` declaration, it produces a page that is too short. In the first case, T<sub>E</sub>X warns you about the extra space by generating an “Underfull `\vbox`” message. With `\raggedbottom` in effect, T<sub>E</sub>X does not warn you about bad page breaks, so you should check your final output for pages that are too short.

The L<sup>A</sup>T<sub>E</sub>X page-breaking commands are analogous to the line-breaking commands described in Section 6.2.1 above. As with line breaking, L<sup>A</sup>T<sub>E</sub>X provides commands to demand or prohibit a page break, with an optional argument transforming the commands to suggestions. The `\pagebreak` and `\nopagebreak` commands are the analogs of `\linebreak` and `\nolinebreak`. When used between paragraphs, they apply to that point; when used in the middle of a paragraph, they apply immediately after the current line. Thus, a `\pagebreak` command within a paragraph insists that T<sub>E</sub>X start a new page after the line in which the command appears, and `\nopagebreak[3]` suggests rather strongly that T<sub>E</sub>X *not* start a new page there.

You will sometimes want to squeeze a little more text on a page than T<sub>E</sub>X thinks you should. The best way of doing this is to make room on the page by removing some vertical space, using the commands of Section 6.4.2. If that doesn't work, you can try using a `\nopagebreak` command to prevent T<sub>E</sub>X from breaking the page before you want it to. However, T<sub>E</sub>X often becomes adamant about breaking a page at a certain point, and it will not be deterred by `\nopagebreak`. When this happens, you can put extra text on the page as follows:

- Add the command `\enlarge this page*{1000pt}` to the text on the current page, before the point where T<sub>E</sub>X wants to start a new page.
- Add a `\pagebreak` command to break the page where you want.

Squeezing in extra text in this way will make the page longer than normal, which may look bad in two-sided printing. Section C.12.2 describes a less heavy-handed approach. But **don't worry about page breaks until you prepare the ultimate, absolutely final, no-more-changes (really!) version.**

The `\newpage` command is the analog of `\newline`, creating a page that ends prematurely right at that point. Even when a `\flushbottom` declaration is in effect, a shortened page is produced. The `\clearpage` command is similar to `\newpage`, except that any leftover figures or tables are put on one or more separate pages with no text. The `\chapter` and `\include` commands (Section 4.4) use `\clearpage` to begin a new page. Adding an extra `\newpage` or `\clearpage` command will not produce a blank page; two such commands in a row are equivalent to a single one. To generate a blank page, you must put some invisible text on it, such as an empty `\mbox`.

When using the `twoside` style option for two-sided printing, you may want to start a sectional unit on a right-hand page. The `\cleardoublepage` command is the same as `\clearpage` except that it produces a blank page if necessary so that the next page will be a right-hand (odd-numbered) one.

When used in two-column format, the `\newpage` and `\pagebreak` commands start a new column rather than a new page. However, the `\clearpage` and `\cleardoublepage` commands start a new page.

The page-breaking commands can be used only where it is possible to start a new page—that is, in paragraph mode and not inside a box (Section 6.4.3). They are all fragile.

## 6.3 Numbering

Every number that L<sup>A</sup>T<sub>E</sub>X generates has a *counter* associated with it. The name of the counter is usually the same as the name of the environment or command that produces the number, except with no `\`. Below is a list of the counters used by L<sup>A</sup>T<sub>E</sub>X's standard document styles to control numbering.

|                            |                           |                         |                      |
|----------------------------|---------------------------|-------------------------|----------------------|
| <code>part</code>          | <code>paragraph</code>    | <code>figure</code>     | <code>enumi</code>   |
| <code>chapter</code>       | <code>subparagraph</code> | <code>table</code>      | <code>enumii</code>  |
| <code>section</code>       | <code>page</code>         | <code>footnote</code>   | <code>enumiii</code> |
| <code>subsection</code>    | <code>equation</code>     | <code>mpfootnote</code> | <code>enumiv</code>  |
| <code>subsubsection</code> |                           |                         |                      |

The counters `enumi` ... `enumiv` control different levels of `enumerate` environments, `enumi` for the outermost level, `enumii` for the next level, and so on. The `mpfootnote` counter numbers footnotes inside a `minipage` environment (Section 6.4.3). In addition to these, an environment created with the `\newtheorem` command (Section 3.4.3) has a counter of the same name unless an optional argument specifies that it is to be numbered the same as another environment.

There are also some other counters used for document-style parameters; they are described in Appendix C.

The value of a counter is a single integer—usually nonnegative. Multiple numbers are generated with separate counters, the “2” and “4” of “Subsection 2.4” coming from the **section** and **subsection** counters, respectively. The value of a counter is initialized to zero and is incremented by the appropriate command or environment. For example, the **subsection** counter is incremented by the `\subsection` command before the subsection number is generated, and it is reset to zero when the **section** counter is incremented, so subsection numbers start from one in a new section.

The `\setcounter` command sets the value of a counter, and `\addtocounter` increments it by a specified amount.

Because<sup>18</sup> counters<sup>17</sup> are stepped before being used, you set them to one less than the number you want.

```
\setcounter{footnote}{17}
Because\footnote{...}
\addtocounter{footnote}{-2}%
counters\footnote{...} are stepped ...
```

When used in the middle of a paragraph, these commands should be attached to a word to avoid adding extra space.

The `\setcounter` and `\addtocounter` commands affect only the specified counter; for example, changing the **section** counter with these commands does not affect the **subsection** counter. The commands to change counter values are global declarations (Section C.1.4); their effects are not limited by the normal scope rules for declarations.

The **page** counter is used to generate the page number. It differs from other counters in that it is incremented *after* the page number is generated, so its value is the number of the current page rather than the next one. A `\setcounter{page}{27}` command in the middle of the document therefore causes the current page to be numbered 27. For this reason, the **page** counter is initialized to one instead of zero.

**LATEX** provides the following commands for printing counter values; the list shows what they produce when the **page** counter has the value four.

|   |                            |    |                           |   |                          |
|---|----------------------------|----|---------------------------|---|--------------------------|
| 4 | <code>\arabic{page}</code> | iv | <code>\roman{page}</code> | d | <code>\alph{page}</code> |
|   |                            | IV | <code>\Roman{page}</code> | D | <code>\Alph{page}</code> |

To generate a printed number, **LATEX** executes a command whose name is formed by adding `\the` to the beginning of the appropriate counter’s name; redefining this command changes the way the number is printed. For example, a subsection number is made by the `\thesubsection` command. To change the numbering of sections and subsections so the fourth subsection of the second section is numbered “II-D”, you type the following (see Section 3.4 for an explanation of `\renewcommand`):

```
\renewcommand{\thesection}{\Roman{section}}
\renewcommand{\thesubsection}{\thesection-\Alph{subsection}}
```

Since sections are usually numbered the same throughout the document (at least until the appendix), the obvious place for this command is in the preamble.

A new counter is created with a `\newcounter` command having the name of the counter as its argument. The new counter's initial value is zero, and its initial `\the...` command prints the value as an arabic numeral. See Section 6.6 for an example of how a new counter is used in defining an environment. The `\newcounter` declaration should be used only in the preamble.

## 6.4 Length, Spaces, and Boxes

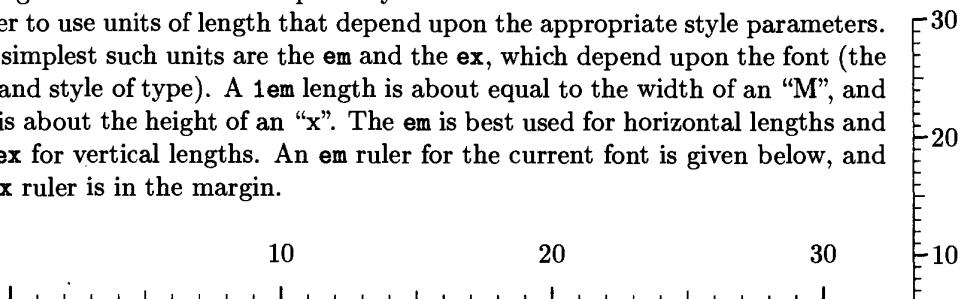
In visual design, one specifies how much vertical space to leave above a chapter heading, how wide a line of text should be, and so on. This section describes the basic tools for making these specifications.

### 6.4.1 Length

A length is a measure of distance. An amount of space or a line width is specified by giving a length as an argument to the appropriate formatting command. A length of one inch is specified by typing `1in`; it can also be given in metric units as `2.54cm` or `25.4mm`, or as `72.27pt`, where `pt` denotes *point*—a unit of length used by printers. A length can also be negative—for example, `-2.54cm`.

The number `0` by itself is not a length. A length of zero is written `0in` or `0cm` or `0pt`, not `0`. Writing `0` as a length is a common mistake.

While inches, centimeters, and points are convenient units, they should be avoided because they specify fixed lengths. A .25-inch horizontal space that looks good in one-column output may be too wide in a two-column format. It's better to use units of length that depend upon the appropriate style parameters. The simplest such units are the `em` and the `ex`, which depend upon the font (the size and style of type). A `1em` length is about equal to the width of an "M", and `1ex` is about the height of an "x". The `em` is best used for horizontal lengths and the `ex` for vertical lengths. An `em` ruler for the current font is given below, and an `ex` ruler is in the margin.



In addition to writing explicit lengths such as `1in` or `3.5em`, you can also express lengths with *length commands*. A length command has a *value* that is a length. For example, `\parindent` is a length command whose value specifies the width of the indentation at the beginning of a normal paragraph. Typing

`\parindent` as the argument of a command is equivalent to typing the current value of `\parindent`. You can also type `2.5\parindent` for a length that is 2.5 times as large as `\parindent`, or `-2.5\parindent` for the negative of that length; `-\parindent` is the same as `-1.0\parindent`.

A length such as `1.5em` or `\parindent` is a *rigid* length. Specifying a space of width `1cm` always produces a one-centimeter-wide space. (It may not be exactly one centimeter wide because your output device might uniformly change all dimensions—for example, enlarging them by 5%.) However, there are also *rubber* lengths that can vary.<sup>2</sup> Space specified with a rubber length can stretch or shrink as required. For example, `TeX` justifies lines (produces an even right margin) by stretching or shrinking the space between words to make each line the same length.

A rubber length has a natural length and a degree of elasticity. Of particular interest is the special length command `\fill` that has a natural length of zero but is infinitely stretchable, so a space of width `\fill` tends to expand as far as it can. The use of such stretchable space is described in Section 6.4.2 below. Multiplying a length command by a number destroys its elasticity, producing a rigid length. Thus, `1\fill` and `.7\fill` are rigid lengths of value zero inches.

Most lengths used in `LATeX` are rigid. Unless a length is explicitly said to be rubber, you can assume it is rigid. All length commands are robust; a `\protect` command should never precede a length command.

Below are some of `LATeX`'s *length parameters*—length commands that define a document's style parameters; others are given in Appendix C. By expressing lengths in terms of these parameters, you can define formatting commands that work properly with different styles.

`\parindent` The amount of indentation at the beginning of a normal paragraph.

`\textwidth` The width of the text on the page.

`\textheight` The height of the body of the page—that is, the normal height of everything on a page excluding the head and foot (Section 6.1.2).

`\parskip` The extra vertical space inserted between paragraphs. It is customary not to leave any extra space between paragraphs, so `\parskip` has a natural length of zero (except in the `letter` document class). However, it is a rubber length, so it can stretch to add vertical space between paragraphs when the `\flushbottom` declaration (Section 6.1.1) is in effect.

`\baselineskip` The normal vertical distance from the bottom of one line to the bottom of the next line in the same paragraph. Thus, `\textheight ÷ \baselineskip` equals the number of lines of text that would appear on a page if it were all one paragraph.<sup>3</sup>

<sup>2</sup>A rigid length is called a `(dimen)` and a rubber length is called a `(skip)` in the `TeXbook`.

<sup>3</sup>This is only approximately correct—see the `\topskip` command in Section C.5.3 to find out why.

**LATEX** provides the following declarations for changing the values of length commands and for creating new ones. These declarations obey the usual scoping rules.

**\newlength** Defines a new length command. You type `\newlength{\gnat}` to make `\gnat` a length command with value `0cm`. An error occurs if a `\gnat` command is already defined.

**\setlength** Sets the value of a length command. The value of `\parskip` is set to `1.01` millimeters by the command `\setlength{\parskip}{1.01mm}`.

**\addtolength** Increments the value of a length command by a specified amount. If the current value of `\parskip` is `.01` inches, then executing the command `\addtolength{\parskip}{-.1\parskip}` changes its value to `.009` inches—the original value plus `-.1` times its original value.

**\settowidth** Sets the value of a length command equal to the width of a specified piece of text. The command `\settowidth{\parindent}{\em small}` sets the value of `\parindent` to the width of *small*—the text produced by typesetting `\em small` in LR mode.

**\settoheight, \settodepth** These commands act like `\settowidth`, except they set the value of the length command to the height and depth, respectively, of the text. For example, `\settoheight{\parskip}{Gnu}` sets the value of `\parskip` to the height of the letter “G” in the current font, while `\settodepth{\parskip}{gnu}` sets the value of `\parskip` to the distance the letter “g” extends below the line. Height and depth are explained below, in Section 6.4.3.

The value of a length command created with `\newlength` can be changed at any time. This is also true for some of **LATEX**’s length parameters, while others should be changed only in the preamble and still others should never be changed. Consult Appendix C to find out when you can safely change the value of a **LATEX** parameter.

## 6.4.2 Spaces

A horizontal space is produced with the `\hspace` command. Think of `\hspace` as making a blank “word”, with spaces before or after it producing an interword space.

Here  is a .5 inch space.  
 Here is a .5 inch space.  
 Here is a .5 inch space.  
 Negative space is a backspace—*like this.*  


Here `\hspace{.5in}` is a .5 inch space.  
 Here `\hspace{.5in}` is a .5 inch space.  
 Here `\hspace{.5in}` is a .5 inch space.  
 ...---like this.`\hspace{-.5in}////`

**T****E****X** removes space from the beginning or end of each line of output text, except at the beginning and end of a paragraph—including space added with `\hspace`. The `\hspace*` command is the same as `\hspace` except that the space it produces is never removed, even when it comes at the beginning or end of a line. The `\hspace` and `\hspace*` commands are robust.

The `\vspace` command produces vertical space. It is most commonly used between paragraphs; when used within a paragraph, the vertical space is added after the line in which the `\vspace` appears.

You seldom add space like this between lines in  
  
 a paragraph, but you sometimes remove space between them by adding some negative space.  


You more often add space between paragraphs—especially before or after displayed material.

`You\vspace{.25in}` seldom add space like this between lines in a paragraph, but you ... by adding some negative space.

`\vspace{7 mm}`

You more often add space between ...

Just as it removes horizontal space from the beginning and end of a line, **T****E****X** removes vertical space that comes at the beginning or end of a page. The `\vspace*` command creates vertical space that is never removed.

If the argument of an `\hspace` or `\vspace` command (or its `*-form`) is a rubber length, the space produced will be able to stretch and shrink. This is normally relevant only for the fine tuning of the formatting commands of a package or document class. However, a space made with an infinitely stretchable length such as `\fill` is useful for positioning text because it stretches as much as it can, pushing everything else aside. The command `\hfill` is an abbreviation for `\hspace{\fill}`.

|           |                  |                                                                  |
|-----------|------------------|------------------------------------------------------------------|
| Here is a | stretched space. | Here is a <code>\hfill</code> stretched space.                   |
| Here are  | two equal ones.  | Here are <code>\hfill</code> two <code>\hfill</code> equal ones. |

Note that when two equally stretchable spaces push against each other, they stretch the same amount. You can use stretchable spaces to center objects or to move them flush against the right-hand margin. However, **L****A****T****E****X** provides more convenient methods of doing that, described in Section 6.5.

Infinitely stretchable space can be used in the analogous way for moving text vertically. The `\vfill` command is equivalent to a blank line followed by `\vspace{\fill}`. Remember that spaces produced by `\hfill` or `\vfill` at the beginning and end of a line or page disappear. You must use `\hspace*{\fill}` or `\vspace*{\fill}` for space that you don't want to disappear.

The `\dotfill` command acts just like `\hfill` except it produces dots instead of spaces. The command `\rulefill` works the same way, but it produces a horizontal line.

Gnats and gnus ..... see pests.  
 This is \_\_\_\_\_ really \_\_\_\_\_ it.

Gnats and gnus \dotfill\ see pests.  
 This is \hrulefill\ really \hrulefill\ it.

### 6.4.3 Boxes

A *box* is a chunk of text that *TeX* treats as a unit, just as if it were a single letter. No matter how big it is, *TeX* will never split a box across lines or across pages. The `\mbox` command introduced in Section 2.2.1 prevents its argument from being split across lines by putting it in a box. Many other *L<sup>A</sup>T<sub>E</sub>X* commands and environments produce boxes. For example, the `array` and `tabular` environments (Section 3.6.2) both produce a single box that can be quite big, as does the `picture` environment described in Section 7.1.

A box has a *reference point*, which is at its left-hand edge. *TeX* produces lines by putting boxes next to one another with their reference points aligned, as shown in Figure 6.1. The figure also shows how the width, height, and depth of a box are computed.

*L<sup>A</sup>T<sub>E</sub>X* provides additional commands and environments for making three kinds of boxes: LR boxes, in which the contents of the box are processed in LR mode; parboxes, in which the contents of the box are processed in paragraph mode; and rule boxes, consisting of a rectangular blob of ink.

A box-making command or environment can be used in any mode. *L<sup>A</sup>T<sub>E</sub>X* uses the declarations in effect at that point when typesetting the box's contents, so the contents of a box appearing in the scope of an `\em` declaration will be emphasized—usually by being set in an italic type style. However, box-making commands that appear in a mathematical formula are not affected by the commands described in Section 3.3.8 that change the type style in the formula.

The `\mbox` in the formula  $x = y * \mathbf{ab} + \mathbf{xyz} + \mathbf{cd} / z$  uses the same type style as the surrounding text.

... in the formula  
 $x = y * \mathbf{ab} +$   
 $\mbox{xyz} + \mathbf{cd} / z$  uses ...

Since the input that produces a box's contents is either the argument of a box-making command or the text of a box-making environment, any declarations made inside it are local to the box.

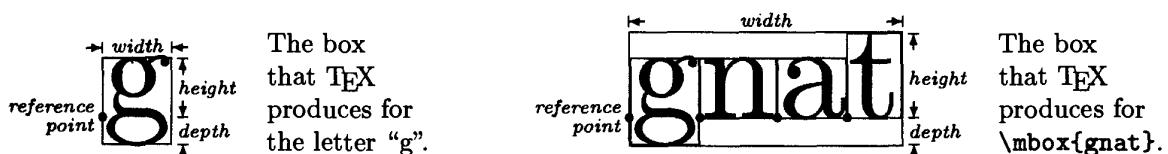


Figure 6.1: Boxes and how *TeX* puts them together.

A box is often displayed on a line by itself. This can be done by treating the box as a formula and using the `displaymath` environment (`\[ ... \]`). The `center` environment described in Section 6.5 can also be used.

### LR Boxes

The `\mbox` command makes an LR box—a box whose contents are obtained by processing the command's argument in LR mode. It is an abbreviated version of the `\makebox` command; `\makebox` has optional arguments that `\mbox` doesn't. The box created by an `\mbox` command is just wide enough to hold its contents. You can specify the width of the box with a `\makebox` command that has an optional first argument. The default is to center the contents in the box, but this can be overridden by a second optional argument that consists of a single letter: `l` to move the contents to the left side of the box, and `r` to move it to the right.

|      |      |      |   |                        |                                                        |
|------|------|------|---|------------------------|--------------------------------------------------------|
| Good | ←    | 1 in | → | are here at last.      | Good <code>\makebox[1in]{\em gnus}</code> are here ... |
| Good | gnus |      |   | are here at last.      | Good <code>\makebox[1in][l]{\em gnus}</code> are ...   |
| Good |      |      |   | gnus are here at last. | Good <code>\makebox[1in][r]{\em gnus}</code> are ...   |

A box is treated just like a word; space characters on either side produce an interword space.

The `\framebox` command is exactly the same as `\makebox` except it puts a frame around the outside of the box. There is also an `\fbox` command, the abbreviation for a `\framebox` command with no optional arguments.

|                 |     |      |   |    |                                                     |
|-----------------|-----|------|---|----|-----------------------------------------------------|
| There was not a | ←   | 1 in | → | or | There was not a <code>\framebox[1in][l]{gnu}</code> |
| There was not a | gnu |      |   |    | or <code>\fbox{armadillo}</code> in sight.          |
| in sight.       |     |      |   |    |                                                     |

When you specify a box of a fixed width,  $\TeX$  acts as if the box has exactly that width. If the contents are too wide for the box, they will overflow into the surrounding text.

X X ~~wide armadillo~~ X X

X X X`\framebox[.5in]{wide armadillo}` X X X

### Parboxes

A *parbox* is a box whose contents are typeset in paragraph mode, with  $\TeX$  producing a series of lines just as in ordinary text. The `figure` and `table` environments (Section 3.5.1) create parboxes. There are two ways to make a parbox at a given point in the text: with the `\parbox` command and the `minipage` environment. They can be used to put one or more paragraphs of text inside a picture or in a table item.

For  $\TeX$  to break text into lines, it must know how wide the lines should be. Therefore, `\parbox` and the `minipage` environment have the width of the parbox

as an argument. The second mandatory argument of the `\parbox` command is the text to be put in the parbox.

← 1 in →  
Breaking lines in  
a narrow parbox YOU CAN  
is hard.

expect to get a lot  
of bad line breaks  
if you try this sort  
of thing.

`\parbox{1in}{Breaking lines in a narrow  
parbox is hard.} \ YOU CAN \  
\parbox{1in}{expect to get a lot of bad  
line breaks if you try ...}`

There is no indentation at the beginning of a paragraph in these parboxes; **LT<sub>E</sub>X** sets the `\parindent` parameter, which specifies the amount of indentation, to zero in a parbox. You can set it to any other value with `\setlength` (Section 6.4.1).

In the example above, the parboxes are positioned vertically so the center of the box is aligned with the center of the text line. An optional first argument of `t` (for *top*) or `b` (*bottom*) aligns the top or bottom line of the parbox with the text line. More precisely, the optional argument causes the reference point of the box to be the reference point of the top or bottom line of its contents.

This is a parbox  
aligned on its bot-  
tom line.

AND THIS one is aligned on  
its top line.

`\parbox[b]{1in}{This is a parbox aligned  
on its bottom line.}  
\ AND THIS \  
\parbox[t]{1in}{one is aligned on its top  
line.}`

Finer control of the vertical positioning is obtained with the `\raisebox` command described below.

The `\parbox` command is generally used for a parbox containing a small amount of text. For a larger parbox or one containing a `tabbing` environment, a list-making environment, or any of the paragraph-making environments described in Section 6.5, you should use a `minipage` environment. The `minipage` environment has the same optional positioning argument and mandatory width argument as the `\parbox` command.

When used in a `minipage` environment, the `\footnote` command puts a footnote at the bottom of the parbox produced by the environment. This is particularly useful for footnotes inside figures or tables. Moreover, unlike in ordinary text, the `\footnote` command can be used anywhere within the environment—even inside another box or in an item of a `tabular` environment. To footnote something in a `minipage` environment with an ordinary footnote at the bottom of the page, use the `\footnotemark` and `\footnotetext` commands described in Section C.3.3.

*gnat*: a tiny bug AND *gnu*: a beast<sup>a</sup> that that is very hard is hard to miss. to find.

<sup>a</sup>See armadillo.

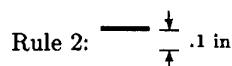
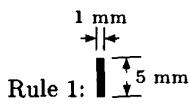
```
\begin{minipage}[t]{1in}
 {\em gnat}: a tiny bug ...
\end{minipage} \ AND \
\begin{minipage}[t]{1in}
 {\em gnu}: a beast\footnote{See
 armadillo.} that is hard to miss.
\end{minipage}
```

If you have one `minipage` environment nested inside another, footnotes may appear at the bottom of the wrong one.

You may find yourself wishing that `TeX` would determine the width of a parbox by itself, making it just wide enough to hold the text inside. This is normally impossible because `TeX` must know the line width to do its line breaking. However, it doesn't have to know a line width when typesetting a tabbing environment because the input specifies where every line ends. Therefore, if a `minipage` environment consists of nothing but a tabbing environment, then `TeX` will set the width of the parbox to be either the width specified by the `minipage` environment's argument or the actual width of the longest line, choosing whichever is smaller.

### Rule Boxes

A rule box is a rectangular blob of ink. It is made with the `\rule` command, whose arguments specify the width and height of the blob. The reference point of the rule box is its lower-left corner. There is also an optional first argument that specifies how high to raise the rule (a negative value lowers it).



Rule 1: `\rule{1mm}{5mm}`  
 Rule 2: `\rule{.1in}{.25in}{.02in}`

A thin enough rule is just a line, so the `\rule` command can draw horizontal or vertical lines of arbitrary length and thickness.

A rule box of width zero is called a *strut*. Having no width, a strut is invisible; but it does have height, and `TeX` will adjust the vertical spacing to leave room for it.

Compare this box with this box.

Compare `\fbox{this box}` with  
`\fbox{\rule{-.5cm}{0cm}{1cm}this box}`.

Struts provide a convenient method of adding vertical space in places where `\vspace` can't be used, such as within a mathematical formula.

## Raising and Lowering Boxes

The `\raisebox` command raises text by a specified length (a negative length lowers the text). It makes an LR box, just like the `\mbox` command.

You can *raise* or *lower* text.

You can `\raisebox{.6ex}{\em raise}` or  
`\raisebox{-.6ex}{\em lower}` text.

It is sometimes useful to change how big  $\TeX$  thinks a piece of text is without changing the text. The `\makebox` command tells  $\TeX$  how wide the text is, while a strut can increase the text's apparent height but cannot decrease it. Optional arguments of `\raisebox` tell  $\TeX$  how tall it should pretend that the text is. The command

```
\raisebox{.4ex}[1.5ex][.75ex]{\em text}
```

not only raises *text* by `.4ex`, but also makes  $\TeX$  think that it extends `1.5ex` above the bottom of the line and `.75ex` below the bottom of the line. (The bottom of the line is where most characters sit; a letter like *y* extends below it.) If you omit the second optional argument,  $\TeX$  will think the text extends as far below the line as it actually does. By changing the apparent height of text, you change how much space  $\TeX$  leaves for it. This is sometimes used to eliminate space above or below a formula or part of a formula.

## Saving Boxes

If a single piece of text appears in several places, you can define a command with `\newcommand` (Section 3.4) to generate it. While this saves typing,  $\TeX$  doesn't save any time because it must do the work of typesetting the text whenever it encounters the command. If the text is complicated—especially if it contains a `picture` environment (Section 7.1)— $\TeX$  could waste a lot of time typesetting it over and over again.

$\TeX$  can typeset something once as a box and then save it in a named *storage bin*, from which it can be used repeatedly. The name of a storage bin is an ordinary command name; a new bin is created and named by the `\newsavebox` declaration. The `\savebox` command makes a box and saves it in a specified bin; it has the bin name as its first argument and the rest of its arguments are the same as for the `\makebox` command. The `\usebox` command prints the contents of a bin.

← .65 in →  
 It's gnats and gnats and gnats , It's `\usebox{\toy}` and `\usebox{\toy}` and  
 wherever we go.

```
\newsavebox{\toy}
\savebox{\toy}[.65in]{gnats}
...
It's \usebox{\toy} and \usebox{\toy} and
\usebox{\toy}, wherever we go.
```

The `\sbox` command is the short form of `\savebox`, with no optional arguments. The `\savebox` and `\sbox` commands are declarations that have the usual scope. However, the `\newsavebox` declaration is global (Section C.1.4) and does not obey the customary scoping rules.

The `lrbox` environment is like an `\sbox` command, except that it ignores spaces at the beginning and end of the text. Thus,

```
\begin{lrbox}{\jewel}
 Text
\end{lrbox}
```

is equivalent to `\sbox{\jewel}{Text}`. This environment is used to define an environment in terms of a command. For example, suppose we want to define a `boxit` environment that produces a boxed, 1-inch-wide paragraph.

Here is a silly boxed paragraph that no one will ever use for anything.

... a \begin{boxit}  
silly boxed paragraph ...  
\end{boxit} for anything.

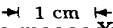
We define this environment by using its body (the text between `\begin{boxit}` and `\end{boxit}`) as the argument of an `\fbox` command. Making the paragraph box with a `minipage` environment, we can define the `boxit` environment by:

```
\newsavebox{\savepar}
\newenvironment{boxit}{\begin{lrbox}{\savepar}
 \begin{minipage}[b]{1in}
 \end{minipage}\end{lrbox}\fbox{\usebox{\savepar}}}
```

#### 6.4.4 Formatting with Boxes

Many L<sup>A</sup>T<sub>E</sub>X users fail to realize how much they can do with the box-making commands described above. I will illustrate the power of these commands with a silly example: defining a `\face` command to put a funny face around a word, like the faces around “funny” and “face” in this sentence. The face is composed of two diamonds (◊) produced with the `\diamond` command, and a smile (¬) produced with `\smile`. The trick, of course, is positioning these symbols correctly relative to the word being “faced”.

The left eye of the face (the one near the “f”) is positioned above the word and 15% of the way from the word’s left edge to its right edge. First, we see how to put a piece of text a fixed distance to the right of the left-hand edge of a word. Recall that if the specified width of a `\makebox` is less than the width of its contents, the contents extend outside the box. With the 1 position specified, the contents extend to the right of the box. In the extreme case of a zero-width box, the entire contents lie to the right.

The gnuuuuu began to moooodoooooo.  ...to \makebox[0pt][1]{\hspace{1cm}X}moo...

Because the box produced by this `\makebox` command has zero width, it doesn't change the position of the "moo...".

We want the face's left eye to be shifted to the right by .15 times the width of the word and raised by the height of the word. So, the `\face` command must measure the width and height of its argument using the `\settoheight` and `\settowidth` commands. These lengths are saved in two new length commands, `\faceht` and `\facewd`, that are defined with `\newlength`. The eye is raised with the `\raisebox` command. Before figuring out how to draw the rest of the face, we define a `\lefteye` command so `\lefteye{\emph{moooo}}` produces .

```
\newlength{\facewd} \newlength{\faceht} %% Define length commands
\newcommand{\lefteye}[1]{% Definition of \lefteye:
 \settowidth{\facewd}{#1}\settoheight{\faceht}{#1}% - Save width & height
 \raisebox{\faceht}{% - Raise by height
 \makebox[0pt][1]{\hspace{.15\facewd}}% - Move right .15 * width
 \diamond}}% - Print eye
#1}% - Print argument
```

Each line in the definition of `\lefteye` ends with a `%`. This both allows comments and splits the definition across lines without introducing unwanted interword spaces. (See Section 2.2.1.) Unintentional spaces are a common source of error in command definitions; be careful not to introduce any.

To make the right eye of the `\face` command, we put immediately after the word a zero-width box whose contents extend to the left. Such a box is produced by a `\makebox` command with the `r` positioning specifier:

```
\makebox[0pt][r]{\diamond\hspace{.15\facewd}}
```

The right eye, like the left, is raised with a `\raisebox` command.

The face's smile is centered beneath the word. To achieve the correct horizontal positioning, `\makebox` is used to center the smile in a box the same width as the word. This box is in turn put inside and extending to the right of a zero-width box that is placed before the word.

```
\makebox[0pt][1]{\makebox[\facewd]{\$smile\$}}
```

A bit of experimentation reveals that the smile should be lowered (raised by a negative distance of) 1.4 ex.

The complete definition of `\face` appears in Figure 6.2. (Like all such definitions, it should go in the preamble.) The definition has been refined to solve two problems. First, the eyes and smile would normally increase the height and depth of the line, causing `TeX` to add extra space above and below it. To prevent this, the optional arguments to `\raisebox` have been used to make `TeX` pretend that the eyes have zero height and the smile has zero (height and) depth.

```

\newlength{\facewd} \newlength{\faceht} %% Define length commands
\newcommand{\face}[1]{%
 \settowidth{\facewd}{#1}\settoheight{\faceht}{#1}%
 \raisebox{\faceht}[0pt][\makebox[0pt][1]{%
 \hspace{.15\facewd}\$\\diamond\$}}% * Save width & height.
 \raisebox{-1.4ex}[0pt][0pt]{\makebox[0pt][1]{%
 \makebox[\facewd]{\$\\smile\$}}}}% * Print left eye.
 \raisebox{\faceht}[0pt][\makebox[0pt][r]{%
 \makebox[\facewd]{\$\\smile\$}}}}% * Print smile. .
 \raisebox{\faceht}[0pt][\makebox[0pt][r]{%
 \$\\diamond\$\\hspace{.15\facewd}}}}% * Print argument
 \raisebox{\faceht}[0pt][\makebox[0pt][r]{%
 \$\\diamond\$\\hspace{.15\facewd}}}}% * Print right eye.
}

```

Figure 6.2: The complete definition of the `\face` command.

The second problem occurs in the definition of `\lefteye` given above. As explained in Section 3.4.1, that definition fails to limit the scope of declarations that appear in its argument.

The `\em` declaration in *gnu escapes its argument.* ... in `\lefteye{\em gnu}` escapes ...

This problem is solved by putting an extra pair of braces around the part of the definition that prints the argument.

A `\face` command uses the argument three times—to measure its height, to measure its width, and to print it. Each time, the argument is processed anew by L<sup>A</sup>T<sub>E</sub>X. This could be a problem. For example, if the argument contains a `\typein` command, each occurrence of `\face` causes L<sup>A</sup>T<sub>E</sub>X to request three separate inputs. We can modify the definition to make `\face` process its argument only once by using a `\savebox` command to save the argument and replacing the three uses (the #1's) with `\usebox` commands.

Solving a formatting problem is often a matter of figuring out how to position some text. The definition of the `\face` command shows how we can use space- and box-making commands to put one object (for example, an eye or a smile) in any desired position relative to another object. The two objects can be arbitrarily far apart. Consider the box around this complete line of text. It is easy to produce the box itself by putting a strut (a zero-width rule) and some horizontal space in an `\fbox` command.

```
\fbox{\rule{0pt}{.5\baselineskip}\hspace{\textwidth}}
```

But, how did I position the box? I could have positioned it relative to the first or last word on the line. However, that would have required knowing where the line breaks occur, and they change every time I revise the paragraph. Instead, I positioned the box relative to an invisible marginal note, produced with an `\mbox{}` inside a `\marginpar` command (described in Section 3.5.2).

Suppose you want to put a box around an entire page of text. Again, producing a box of the right size is easy; the problem is positioning it. Since the

box appears at a fixed point on the page, it must be positioned relative to some object that appears at a fixed location on the page. One such object is the page heading; another is the page number. You just have to add a suitable zero-width box to either of these objects. Section C.5.3 describes how to set page headings; the page number is produced by the `\thepage` command (Section 6.3).

## 6.5 Centering and “Flushing”

The `center` environment is used to produce one or more lines of centered text; a `\\"` command starts a new line.

This is the last line of text in the preceding paragraph.

Here are three  
centered  
lines of text.

... of text in the preceding paragraph.  
`\begin{center}`  
 Here are three\\ centered \\  
 lines of text.  
`\end{center}`  
 This is the text immediately ...

This is the text immediately following the environment. It begins a new paragraph only if you leave a blank line after the `\end{center}`.

`LaTeX` is in paragraph mode inside the `center` environment, so it breaks lines where necessary to keep them from extending past the margins.

The `flushleft` and `flushright` environments are similar, except instead of each line of text being centered, it is moved to the left or right margin, respectively.

These are the last lines of text from the preceding paragraph.

These are two  
flushed right lines.

... of text from the preceding paragraph.  
`\begin{flushright}`  
 These are two \\ flushed right lines.  
`\end{flushright}`

The `center` and `flushright` environments are most commonly used with each new line started by an explicit `\\"` command. There is little purpose to using the `flushleft` environment in this way, since the `\\"` command in ordinary text produces a flushed-left line. By letting `TeX` do the line breaking, `flushleft` produces ragged-right text.

Notice how `TeX` leaves these lines uneven, without stretching them out to reach the right margin. This is known as “ragged-right” text.

`\begin{flushleft}`  
 Notice how \TeX\ leaves these lines  
 uneven, without stretching them out ...  
`\end{flushleft}`

The centering and flushing environments work by using certain declarations that change how `TeX` makes paragraphs. These declarations are available as

*L**A**T**E**X* commands; the declaration that corresponds to each environment is shown below:

|                     |                         |                          |                           |
|---------------------|-------------------------|--------------------------|---------------------------|
| <i>environment:</i> | <code>center</code>     | <code>flushleft</code>   | <code>flushright</code>   |
| <i>declaration:</i> | <code>\centering</code> | <code>\raggedleft</code> | <code>\raggedright</code> |

These declarations can be used inside an environment such as `quote` or in a `parbox` (Section 6.4.3).

This is text that comes at the end of the preceding paragraph.

Here is a quote environment  
whose lines are  
flushed right.

... at the end of the preceding paragraph.  
`\begin{quote}`  
`\raggedleft` Here is a quote environment\\  
 whose lines are \\ flushed right.  
`\end{quote}`

The text of a figure or table can be centered on the page by putting a `\centering` declaration at the beginning of the `figure` or `table` environment.

Unlike the environments, the centering and flushing declarations do not start a new paragraph; they simply change how *TeX* formats paragraph units (Section 6.2.1). To affect a paragraph unit's format, the scope of the declaration must contain the blank line or `\end` command (of an environment like `quote`) that ends the paragraph unit.

## 6.6 List-Making Environments

A *list* is a sequence of items typeset in paragraph mode with indented left and right margins, each item begun with a label. A label can be empty and an indentation can be of length zero, so an environment not normally thought of as a list can be regarded as one. In fact, almost every one of *L**A**T**E**X*'s environments that begins on a new line is defined as a list. The list-making environments are: `quote`, `quotation`, `verse`, `itemize`, `enumerate`, `description`, `thebibliography`, `center`, `flushleft`, and `flushright`, as well as the theorem-like environments declared by `\newtheorem`.

*L**A**T**E**X* provides two primitive list-making environments: `list` and `trivlist`, the latter being a restricted version of `list`. They are flexible enough to produce most lists and are used to define the environments listed above.

### 6.6.1 The `list` Environment

The `list` environment has two arguments. The first specifies how items should be labeled when no argument is given to the `\item` command; the second contains declarations to set the formatting parameters. The general form of a list and the meaning of most of its formatting parameters are shown in Figure 6.3. The vertical-space parameters are rubber lengths; the horizontal-space param-

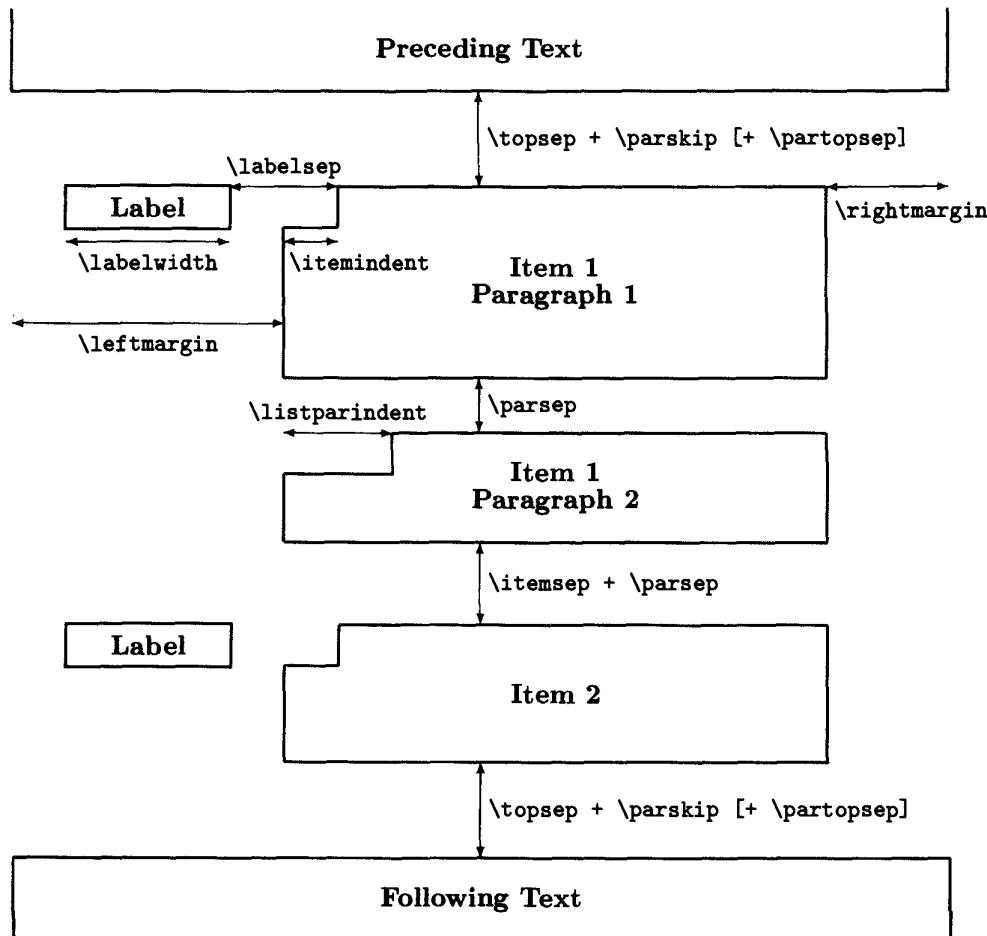


Figure 6.3: The format of a list.

ters are rigid ones. The extra `\partopsep` space is added at the top of the list only if the input file has a blank line before the environment. The vertical space following the environment is the same as the one preceding it.

Inside the list, the values of `\parskip` and `\parindent` are set to the values of `\parsep` and `\listparindent`, respectively. When one list is nested inside another, the `\leftmargin` and `\rightmargin` distances of the inner list are measured from the margins of the outer list.

The default values of these parameters are determined by the document class (and its options), as described in Section C.6.3; they depend upon the level of nesting of the list. These default values can be changed by declarations in the `list` environment's second argument. It is best to maintain the same spacing in all lists, so the default values of the vertical spacing and margin parameters should be used. However, the width and placement of the label may differ in different kinds of lists.

The label is typeset in LR mode. If it fits within a box of width `\labelwidth`, it is placed flush with the right-hand edge of a box of that width, which is positioned as shown in Figure 6.3. (It can be moved to a different position with the `\hfill` command of Section 6.4.2.) If the label is wider than `\labelwidth`, it extends to the right of the position shown in Figure 6.3 and the indentation of the first line of the item is increased by the extra width of the label.

The first argument of the environment is the text to be used as the label for any `\item` command with no optional argument. To number the items automatically, the second argument of the `list` environment should contain a `\usecounter{ctr}` command whose argument is the name of a counter—usually one defined with `\newcounter` (Section 6.3). This counter is reset to zero at the beginning of the environment and is incremented by one before the execution of any `\item` command that has no optional argument, so it can be used to generate a label number.

This sentence represents the end of the text that precedes the list.

B-I This is the first item of the list. Observe how the left and right margins are indented by the same amount.

B-II This is the second item.

As usual, the following text starts a new paragraph only if the `list` environment is followed by a blank line.

A `list` environment like this would be used to produce a one-of-a-kind list. The `list` environment is more commonly used with the `\newenvironment` command (Section 3.4) to define a new environment. Having many different list formats tends to confuse the reader. Instead of formatting each list individually, you should define a small number of list-making environments.

```
\newcounter{bean}
... the text that precedes the list.
\begin{list}
{B--\Roman{bean}}{\usecounter{bean}}
\setlength{\rightmargin}{\leftmargin}
\item This is the first item of the list.
Observe how the left and ...
\item This is the second item.
\end{list}
As usual, the following text starts a ...
```

### 6.6.2 The `trivlist` Environment

The `trivlist` environment is a restricted form of the `list` environment in which margins are not indented and an `\item` command with no optional argument produces no text. The environment has no arguments and is very much like a `list` environment whose second argument sets `\leftmargin`, `\rightmargin`, `\labelwidth`, and `\itemindent` to a length of zero.

The `trivlist` environment is used to define other environments that create a one-item list, usually with an empty label. For example, the `center` environment (Section 6.5) is equivalent to

```
\begin{trivlist} \centering \item ... \end{trivlist}
```

## 6.7 Fonts

A *font* is a particular size and style of type. Section 3.1 explains how to change the type style; Section 6.7.1 explains how to change the size. As its default, L<sup>A</sup>T<sub>E</sub>X uses the Computer Modern fonts designed by Donald Knuth. There are a variety of packages that cause L<sup>A</sup>T<sub>E</sub>X to use fonts other than Computer Modern. Many of these packages are described in the *L<sup>A</sup>T<sub>E</sub>X Companion*; your *Local Guide* will tell you which of them are available on your computer. You should use one of these packages to change the fonts that are used throughout the document.

### 6.7.1 Changing Type Size

L<sup>A</sup>T<sub>E</sub>X's normal default type size is ten-point, but the `11pt` document-class option makes the default size eleven-point and the `12pt` option makes it twelve-point. L<sup>A</sup>T<sub>E</sub>X provides the following declarations for changing the type size within a document.

|     |                            |     |                          |     |                     |
|-----|----------------------------|-----|--------------------------|-----|---------------------|
| Gnu | <code>\tiny</code>         | Gnu | <code>\normalsize</code> | Gnu | <code>\LARGE</code> |
| Gnu | <code>\scriptsize</code>   | Gnu | <code>\large</code>      | Gnu | <code>\huge</code>  |
| Gnu | <code>\footnotesize</code> | Gnu | <code>\Large</code>      | Gnu | <code>\Huge</code>  |
| Gnu | <code>\small</code>        |     |                          |     |                     |

These declarations can be combined in the natural way with the commands for changing the type style described in Section 3.1.

**get big and bigger and bolder**

`\sffamily` get `\large` `\big` and `\Large` `\bigg`  
`\bfseries` and `\LARGE` `\boldsymbol`

The precise size of type produced by these declarations depends upon the default type size; the examples appearing here are for a ten-point default size. The `\normalsize` declaration produces the default size, `\footnotesize` produces the

size used for footnotes, and `\scriptsize` produces the size used for subscripts and superscripts in `\normalsize` formulas.

When you typeset an entire paragraph unit (Section 6.2.1) in a certain size, the scope of the size-changing declaration should include the blank line or `\end` command that ends the paragraph unit. A size-changing command may not be used in math mode. To set part of a formula in a different size of type, you can put it in an `\mbox` containing the size-changing command. All size-changing commands are fragile.

Not every type style is available in every size. If you try to use a font that is not available, L<sup>A</sup>T<sub>E</sub>X will issue a warning and substitute a font of the same size that is as close as possible in style to the one you wanted.

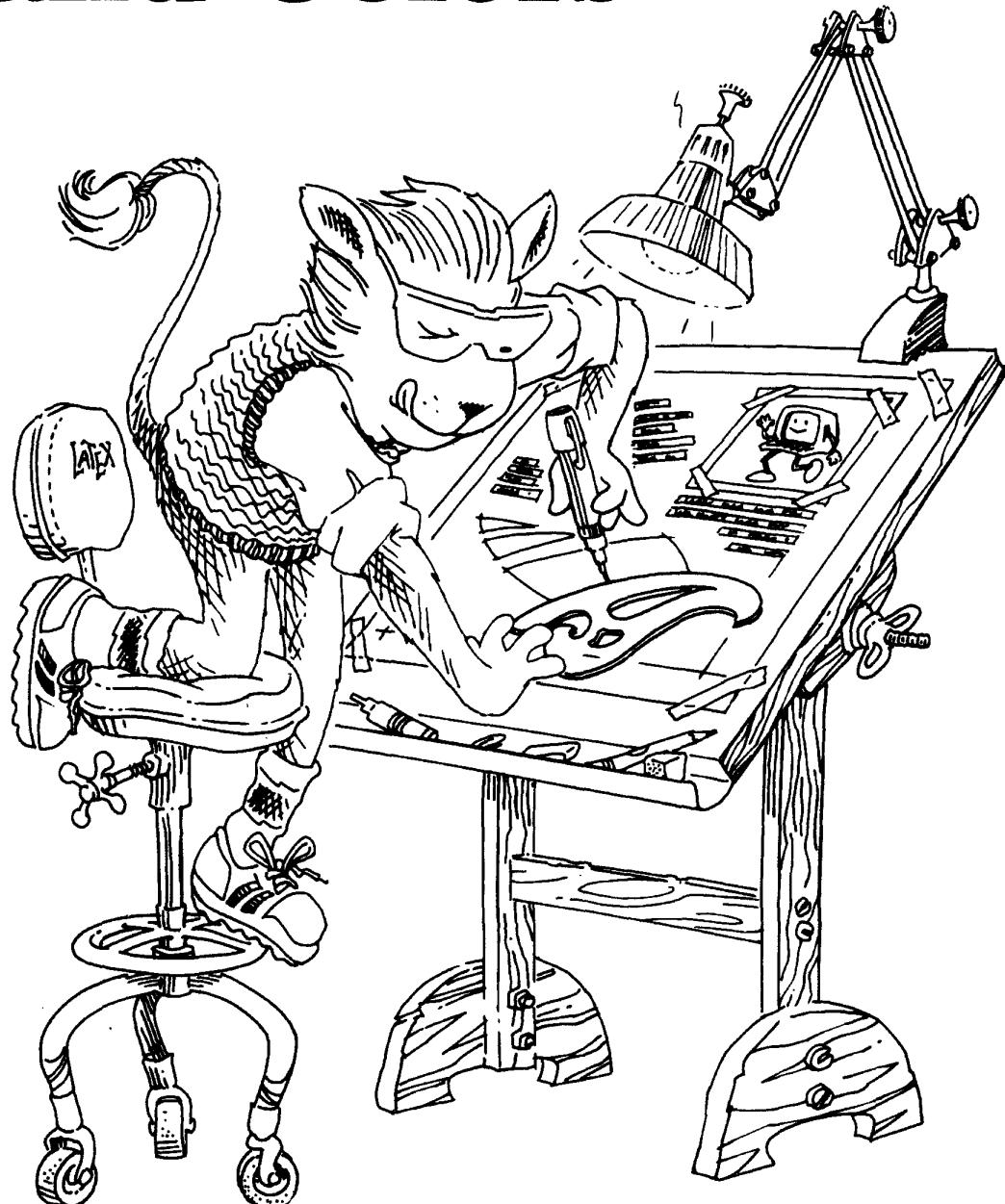
### 6.7.2 Special Symbols

You may need a special symbol not normally provided by L<sup>A</sup>T<sub>E</sub>X. Such symbols can exist in special fonts. The *Local Guide* should tell you what special fonts are available; the *L<sup>A</sup>T<sub>E</sub>X Companion* describes how to get L<sup>A</sup>T<sub>E</sub>X to use a special font.

A symbol in a special font is often identified by its *character-code*, which is a number from 0 to 255. When describing a special font, the *Local Guide* should tell you how to find the character codes for its special symbols. The command `\symbol{26}` produces the symbol with character code 26 in the currently chosen font. Character codes are often given in octal (base 8) or hexadecimal (base 16). An octal character code is prefaced by ' and a hexadecimal one by ", so `\symbol{'32}` and `\symbol{"1A}` produce the same symbol as `\symbol{26}`, since 32 is the octal and 1A the hexadecimal representation of 26.

## CHAPTER 7

# Pictures and Colors



This chapter explains how to draw pictures, include graphics prepared with other programs in your document, and produce colors. Of these features, only simple picture-drawing commands are implemented in standard  $\text{\TeX}$ . The others are implemented by the `graphics` and `color` packages, which need special help from the device driver—the program that turns the `dvi` file into the output you see. Different device drivers may require different `dvi` files, which are created by giving different options to the `\usepackage` commands that load the `graphics` and `color` packages.

Eventually, these special features should be available with all device drivers. However, when  $\text{\LaTeX}\ 2\epsilon$  is first released, some drivers will not implement these features, and some drivers may implement only some of them. Check the *Local Guide* to find out about the device drivers on your computer.

## 7.1 Pictures

The `picture` environment is used to draw pictures composed of text, straight lines, arrows, circles, and simple curves. This environment and its associated commands are implemented with standard  $\text{\TeX}$ ; they do not require special support from the device driver. However, the `pict2e` package uses device-driver support to provide enhanced versions of these commands that remove some of their restrictions. The enhanced commands can draw straight lines and arrows of any slope, circles of any size, and lines (straight and curved) of any thickness.

In the `picture` environment, you position objects in a picture by specifying their *x* and *y* coordinates. So, before getting to the picture-making commands, let us first review a little bit of coordinate geometry.

A *coordinate* is a number such as 5, -7, 2.3, or -3.1416. Given an *origin* and a *unit length*, a pair of coordinates specifies a position. As shown in Figure 7.1, the coordinate pair (-1.8, 1) specifies the position reached by starting at the origin and moving left 1.8 units and up 1 unit.

The *unit length* used in determining positions in a `picture` environment is the value of the `\unitlength` command. Not just positions but all lengths in a `picture` environment are specified in terms of `\unitlength`. Its default value is 1 point (about 1/72<sup>nd</sup> of an inch or .35mm), but it can be changed with the `\setlength` command described in Section 6.4.1. Changing the value of `\unitlength` magnifies or reduces a picture—halving the value halves the lengths of all lines and the diameters of all circles. This makes it easy to adjust the size of a picture. However, changing `\unitlength` does not change the widths of lines or the size of text characters, so it does not provide true magnification and reduction. (The `graphics` package, described in Section 7.2, provides true magnifying and reducing commands.)

$\text{\LaTeX}$  provides two standard thicknesses for the lines in a picture—thin as in  and thick as in . They are specified by the declarations `\thinlines`

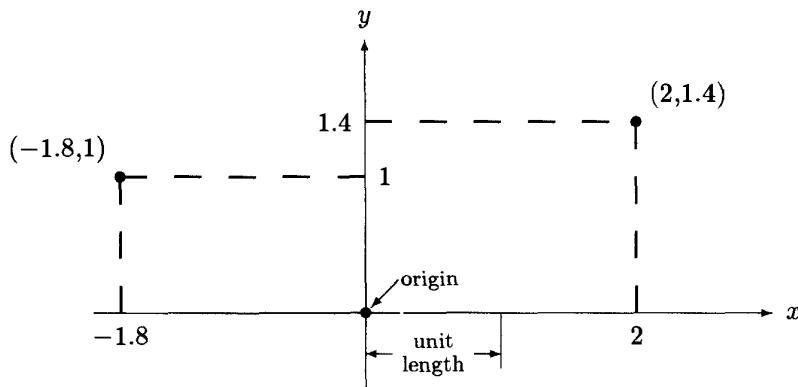


Figure 7.1: Points and their coordinates.

and `\thicklines`, with `\thinlines` as the default. These commands are ordinary declarations and can be used at any time.

Many picture-drawing commands have a coordinate pair as an argument. Such an argument is not enclosed in braces, but is just typed with parentheses and a comma, as in `(-2,3.7)` or `(0,-17.2)`.

### 7.1.1 The picture Environment

The `picture` environment has a coordinate-pair argument that specifies the picture's size (in terms of `\unitlength`). The environment produces a box (Section 6.4.3) whose width and height are given by the two coordinates. The origin's default position is the lower-left corner of this box. However, the `picture` environment has an optional second coordinate-pair argument that specifies the coordinates of the box's lower-left corner, thereby determining the position of the origin. For example, the command

```
\begin{picture}(100,200)(10,20)
```

produces a picture of width 100 units and height 200 units, whose lower-left corner has coordinates `(10, 20)`, so the upper-right corner has coordinates `(110, 220)`. Unlike ordinary optional arguments, the `picture` environment's optional argument is not enclosed in square brackets.

When first drawing a picture, you will usually omit the optional argument, leaving the origin at the lower-left corner. Later, if you want to modify the picture by shifting everything, you just add the appropriate optional argument.

The environment's first argument specifies the nominal size of the picture, which is used by TeX to determine how much room to leave for it. This need

bear no relation to how large the picture really is; L<sup>A</sup>T<sub>E</sub>X allows you to draw things outside the picture, or even off the page.

The `\begin{picture}` command puts L<sup>A</sup>T<sub>E</sub>X in *picture mode*, a special mode that occurs nowhere else.<sup>1</sup> The only things that can appear in picture mode are `\put`, `\multiput`, `\qbezier`, and `\graphpaper` commands (described below) and declarations such as `\em`, `\thicklines`, and `\setlength`. You should not change `\unitlength` in picture mode.

The examples in this section all illustrate commands in picture mode, but the `\begin{picture}` and `\end{picture}` commands are not shown. To help you think in terms of arbitrary unit lengths, the examples assume different values of `\unitlength`. They are all drawn with the `\thicklines` declaration in effect. The pictures in the examples also contain lines and arrows, not produced by the commands being illustrated, that indicate positions and dimensions; these are drawn with `\thinlines` in effect, allowing you to compare the two line thicknesses.

Remember that the `picture` environment produces a box, which T<sub>E</sub>X treats just like a single (usually) large letter. See Section 6.5 for commands and environments to position the entire picture on the page. All the picture-drawing commands described in this section are fragile.

### 7.1.2 Picture Objects

Most things in a picture are drawn by the `\put` command. The command

```
\put (11.3,-.3){picture object}
```

puts the *picture object* in the picture with its *reference point* having coordinates  $(11.3, -0.3)$ . The various kinds of picture objects and their reference points are described below.

#### Text

The simplest kind of picture object is ordinary text, which is typeset in LR mode to produce a box with the usual reference point (see Section 6.4.3).

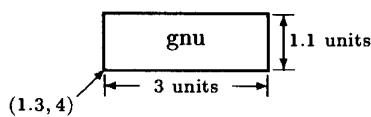
gang of armadillos  
(2.3, 5) 

#### Boxes

A box picture object is made with the `\makebox` or `\framebox` command. These commands, and the related `\savebox` command, have a special form for use with pictures. The first argument is a coordinate pair that specifies the width and height of the box.

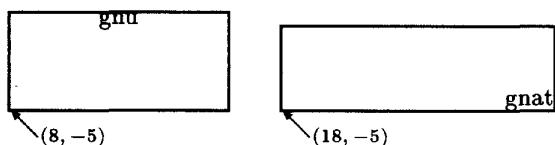
---

<sup>1</sup>L<sup>A</sup>T<sub>E</sub>X's picture mode is really a restricted form of LR mode.



```
\put(1.3,4){\framebox(3,1.1){gnu}}
```

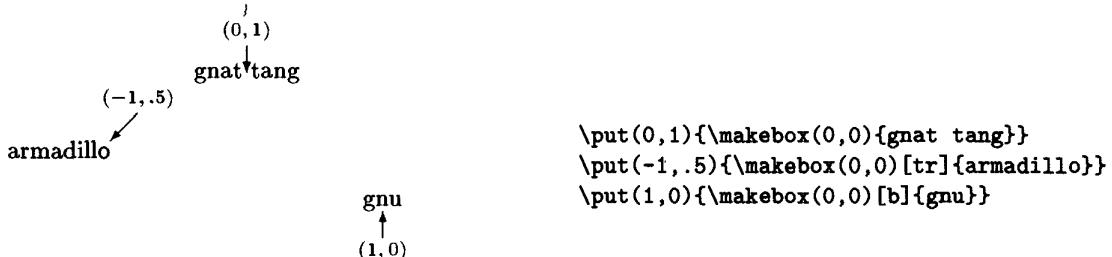
The reference point is the lower-left corner of the box. The default is to center the text both horizontally and vertically within the box, but an optional argument specifies other positioning. This argument consists of one or two of the following letters: **l** (left), **r** (right), **t** (top), and **b** (bottom). The letters in a two-letter argument can appear in either order.



```
\put(8,-5){\framebox(8,3.5)[t]{gnu}}
\put(18,-5){\framebox(10,3)[br]{gnat}}
```

Unlike the ordinary `\framebox` command described in Section 6.4.3, the picture-making version adds no space between the frame and the text. There is a corresponding version of `\makebox` that works the same as `\framebox` except it does not draw the frame. These picture-making versions are used mainly as picture objects, although they can be used anywhere that an ordinary box-making command can.

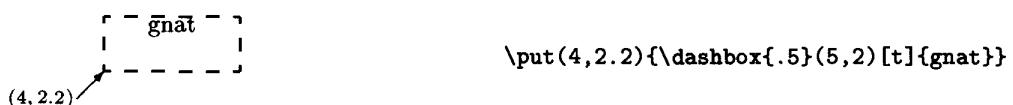
The discussion of zero-width boxes in Section 6.4.4 should explain why a `\makebox(0,0)` command with no positioning argument puts the center of the text on the reference point, and with a positioning argument puts the indicated edge or corner of the text on the reference point.



```
\put(0,1){\makebox(0,0){gnat tang}}
\put(-1,.5){\makebox(0,0)[tr]{armadillo}}
\put(1,0){\makebox(0,0)[b]{gnu}}
```

A `\makebox(0,0)` command is very useful for positioning text in a picture.

The `\dashbox` command is similar to `\framebox` but draws the frame with dashed lines. It has an additional first argument that specifies the width of each dash.



```
\put(4,2.2){\dashbox{.5}(5,2)[t]{gnat}}
```

A dashed box looks best when the width and the height are both multiples of the dash length—in this example, the width is ten times and the height four times the length of a dash.

### Straight Lines

Straight lines can be drawn with only a fixed, though fairly large, choice of slopes. A line is not specified by giving its endpoints, since that might produce a slope not in L<sup>A</sup>T<sub>E</sub>X's repertoire. Instead, the slope and length of the line are specified. L<sup>A</sup>T<sub>E</sub>X's method of describing slope and length was chosen to make designing pictures easier, but it requires a bit of explanation.

The `\line` command produces a picture object that is a straight line, with one end of the line as its reference point. The command has the form

`\line(x,y){len}`

where the coordinate pair  $(x, y)$  specifies the slope and  $len$  specifies the length, in a manner I will now describe. (Figure 7.2 illustrates the following explanation with a particular example.) Let  $p_0$  be the reference point, and suppose its coordinates are  $(x_0, y_0)$ . Starting at  $p_0$ , move  $x$  units to the right and  $y$  units up to find the point  $p_1$ , so  $p_1$  has coordinates  $(x_0 + x, y_0 + y)$ . (Negative distances have the expected meaning: moving right a distance of  $-2$  units means moving 2 units to the left, and moving up  $-2$  units means moving down 2 units.) The line drawn by this command lies along the straight line through  $p_0$  and  $p_1$ . It starts at  $p_0$  and goes in the direction of  $p_1$  a distance determined as follows by  $len$ . If the line is not vertical ( $x \neq 0$ ), it extends  $len$  units horizontally to the right or left of  $p_0$  (depending upon whether  $x$  is positive or negative). If the line is vertical ( $x = 0$ ), it extends  $len$  units above or below  $p_0$  (depending upon whether  $y$  is positive or negative).

The  $len$  argument therefore specifies the line's horizontal extent, except for a vertical line, which has no horizontal extent, where it specifies the vertical

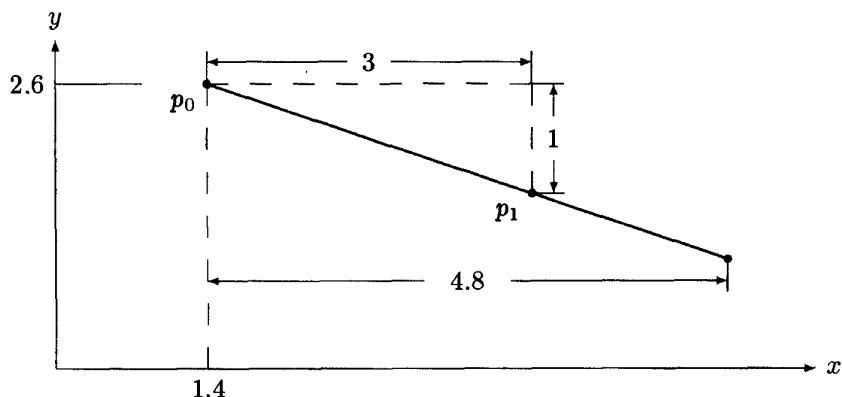


Figure 7.2: `\put (1.4,2.6){\line(3,-1){4.8}}`

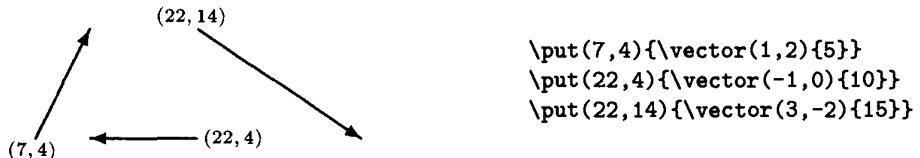
distance. It equals the actual length of the line only for horizontal and vertical lines. The value of *len* must be nonnegative.

Since only a fixed number of slopes are available, there are only a limited number of values that *x* and *y* can assume. They must both be integers (numbers without decimal points) between  $-6$  and  $+6$ , inclusive. Moreover, they can have no common divisor bigger than one. In other words,  $x/y$  must be a fraction in its simplest form, so you can't let  $x = 2$  and  $y = -4$ ; you must use  $x = 1$  and  $y = -2$  instead. The following are all *illegal* arguments of a `\line` command:  $(1.4, 3)$ ,  $(3, 6)$ ,  $(0, 2)$ , and  $(1, 7)$ .

*L<sup>A</sup>T<sub>E</sub>X* draws slanted (neither horizontal nor vertical) lines using a special font whose characters consist of small line segments. This means that there is a smallest slanted line that *L<sup>A</sup>T<sub>E</sub>X* can draw—its length is about 10 points, or  $1/7$  inch. If you try to draw a smaller slanted line, *L<sup>A</sup>T<sub>E</sub>X* will print nothing. It also means that *L<sup>A</sup>T<sub>E</sub>X* must print lots of line segments to make up a long slanted line, so drawing a lot of slanted lines can take a long time and can use up a lot of *T<sub>E</sub>X*'s memory. However, *L<sup>A</sup>T<sub>E</sub>X* draws a horizontal or vertical line of any length quickly, without using much memory.

## Arrows

An arrow—a straight line ending in an arrowhead—is made by the `\vector` command. It works exactly like the `\line` command.



The tip of the arrowhead lies on the endpoint of the line opposite the reference point. This makes any normal-length arrow point away from the reference point. However, for an arrow of length zero, both endpoints lie on the reference point, so the tip of the arrow is at the reference point.

*L<sup>A</sup>T<sub>E</sub>X* can't draw arrows with as many different slopes as it can draw lines. The pair of integers specifying the slope in a `\vector` command must lie between  $-4$  and  $+4$ , inclusive; as with the `\line` command, they must have no common divisor.

## Stacks

The `\shortstack` command produces a box containing a single column of text with the reference point at its lower-left corner. Its argument contains the text, rows being separated by a `\backslash\backslash` command. The `\shortstack` command is much like a one-column `tabular` environment (Section 3.6.2), but the space between rows is designed for a vertical column of text in a picture. The default alignment

is to center each row in the column, but an optional positioning argument of `l` (left) or `r` (right) aligns the text on the indicated edge.

```
\put(1,7){\shortstack{Gnats\\ and\\ gnus}}
\put(3,7){\shortstack[r]{May\\ break\\ my}}
\put(5,7){\shortstack{l}{Sh\\ o\\ e\\ s}}
```

Unlike an ordinary `tabular` environment, rows are not evenly spaced. You can change the inter-row spacing by using either the `\backslash` command's optional argument (Section C.1.6) or a strut (Section 6.4.3). The `\shortstack` command is an ordinary box-making command that can be used anywhere, but it seldom appears outside a `picture` environment.

### Circles

The `\circle` command draws a circle of the indicated diameter, with the center of the circle as reference point, and the `\circle*` command draws a disk (a circle with the center filled in). L<sup>A</sup>T<sub>E</sub>X has only a fixed collection of circles and disks; the `\circle` and `\circle*` commands choose the one whose diameter is closest to the specified diameter.

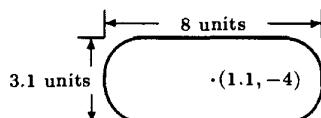


```
\put(20,0){\circle{20}}
\put(20,0){\vector(0,1){10}}
\put(50,0){\circle*{5}}
```

On my computer, the largest circle that L<sup>A</sup>T<sub>E</sub>X can draw has a diameter of 40 points (a little more than 1/2 inch) and the largest disk has a diameter of 15 points (about .2 inch). Consult the *Local Guide* to find out what size circles and disks are available on your computer.

### Ovals and Rounded Corners

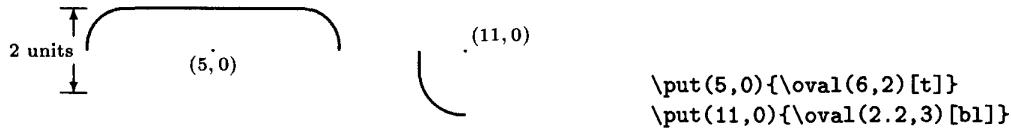
An oval is a rectangle with rounded corners—that is, a rectangle whose corners are replaced by quarter circles. It is generated with the `\oval` command, whose argument specifies the width and height, the reference point being the center of the oval. L<sup>A</sup>T<sub>E</sub>X draws the oval with corners as round as possible, using quarter circles with the largest possible radius.



```
\put(1.1,-4){\oval(8,3.1)}
```

Giving an optional argument to the `\oval` command causes L<sup>A</sup>T<sub>E</sub>X to draw only half or a quarter of the complete oval. The argument is one or two of the letters `l` (left), `r` (right), `t` (top), and `b` (bottom), a one-letter argument specifying

a half oval and a two-letter argument specifying a quarter oval. The size and reference point are determined as if the complete oval were being drawn; the optional argument serves only to suppress the unwanted part.

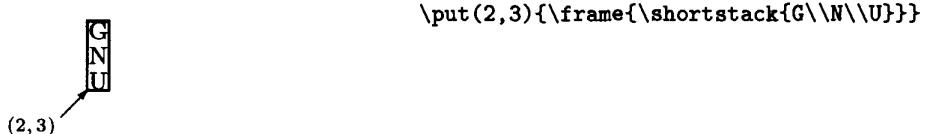


Joining a quarter oval to straight lines produces a rounded corner. It takes a bit of calculating to figure out where to \put the quarter oval.



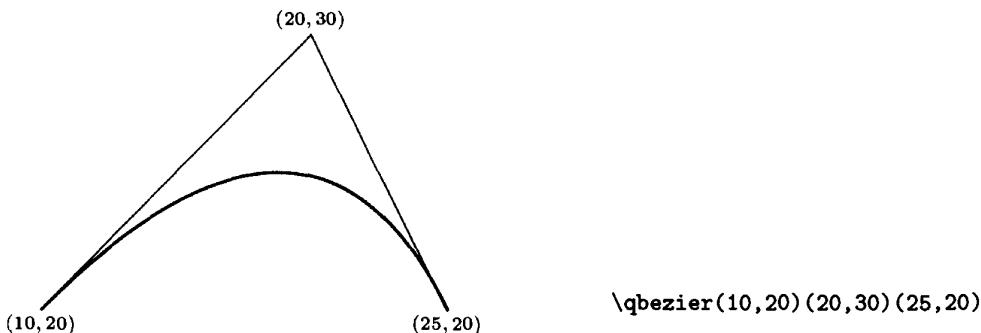
## Framing

The `\framebox` command puts a frame of a specified size around an object. It is often convenient to let the size of the object determine the size of the frame. The `\fbox` command described in Section 6.4.3 does this, but it puts extra space around the object that you may not want in a picture. The `\frame` command works very much like `\fbox` except it doesn't add any extra space.



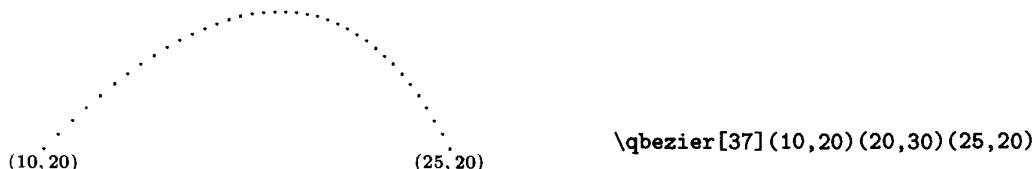
### 7.1.3 Curves

The `\qbezier` command takes three points as arguments and draws a quadratic Bezier curve with those as its control points. A quadratic Bezier curve with control points  $P_1$ ,  $P_2$ ,  $P_3$  is a curve from  $P_1$  to  $P_3$  such that the line from  $P_1$  to  $P_2$  is tangent to the curve at  $P_1$ , and the line from  $P_3$  to  $P_2$  is tangent to the curve at  $P_3$ .



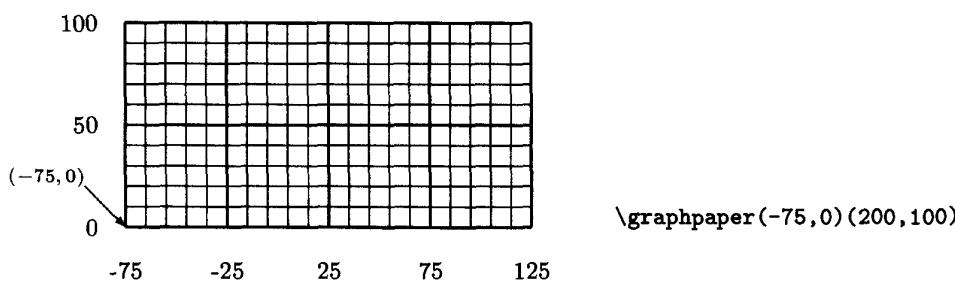
For two curves (or a curve and a straight line) to join smoothly, they must have the same tangent at the point where they meet. Bezier control points therefore provide a convenient way of specifying a curve.

$\text{\LaTeX}$  draws curves by drawing lots of individual points. (This does not apply to the `pict2e` package's enhanced version.) Each point takes time to draw and takes memory space, so  $\text{\LaTeX}$  does not draw a completely smooth curve. The `\qbeziermax` command specifies the maximum number of points  $\text{\LaTeX}$  will normally plot for a single `\qbezier` command. (It can be changed with `\renewcommand`.) However, an optional argument to `\qbezier` allows you to tell  $\text{\LaTeX}$  exactly how many points to plot.

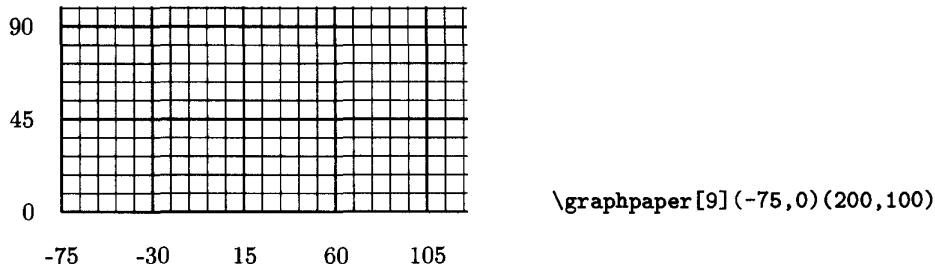


#### 7.1.4 Grids

The `graphpap` package defines the `graphpaper` command, which draws a numbered coordinate grid. The command's first argument specifies the coordinates of the lower-left corner of the grid, and its second argument specifies the grid's width and height.



The grid normally consists of one line every 10 units, but an optional first argument allows you to specify a different spacing.



The arguments of `\graphpaper` can contain only integers.

### 7.1.5 Reusing Objects

The `\savebox` command described in Section 6.4.3 is similar to `\makebox` except that, instead of being drawn, the box is saved in the indicated storage bin. Like `\makebox`, the `\savebox` command has a form in which the size of the box is indicated by a coordinate pair, with positioning determined by an optional argument.



The storage bin `\toy` in this example must be defined with `\newsavebox`. A `\savebox` command can be used inside a `picture` environment to save an object that appears several times in that picture, or outside to save an object that appears in more than one picture. Remember that `\savebox` is a declaration with the normal scoping rules.

It can take L<sup>A</sup>T<sub>E</sub>X a long time to draw a picture, especially if it contains slanted lines, so it's a good idea to use `\savebox` whenever an object appears in different pictures or in different places within the same picture. However, a saved box also uses memory, so a picture should be saved no longer than necessary. The space used by a saved box is reclaimed upon leaving the scope of the `\savebox` declaration. You can also use a command like `\sbox{\toy}{}{}`, which destroys the contents of storage bin `\toy` and reclaims its space.

### 7.1.6 Repeated Patterns

Pictures often contain repeated patterns formed by regularly spaced copies of the same object. Instead of using a sequence of `\put` commands, such a pattern

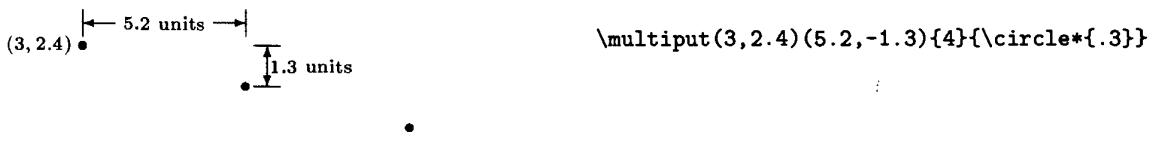
can be drawn with a `\multiput` command. For any coordinate pairs  $(x, y)$  and  $(\Delta x, \Delta y)$ , the command

```
\multiput(x,y)(\Delta x,\Delta y){17}{object}
```

puts 17 copies of *object* in the picture, starting at position  $(x, y)$  and stepping the position by  $(\Delta x, \Delta y)$  units each time. It is equivalent to the 17 commands

```
\put(x,y){object}
\put(x+\Delta x,y+\Delta y){object}
\put(x+2\Delta x,y+2\Delta y){object}
...
\put(x+16\Delta x,y+16\Delta y){object}
```

as illustrated by the following example:



You can make a two-dimensional pattern by using a `picture` environment containing another `\multiput` in the argument of a `\multiput` command. However, `\multiput` typesets the object anew for each copy it makes, so it is much more efficient to make a two-dimensional pattern by saving a one-dimensional pattern made with `\multiput` in a storage bin, then repeating it with another `\multiput`. Saving the object in a bin can also save processing time for a one-dimensional pattern. However, patterns with too many repetitions in all may cause TeX to run out of memory.

### 7.1.7 Some Hints on Drawing Pictures

As you gain experience with the `picture` environment, you'll develop your own techniques for designing pictures. Here are a few hints to get you started.

If you use a small unit length, such as the default value of 1 point, you will seldom need decimals.

It can take quite a bit of trial and error to get a picture right. Use a screen previewer, and run L<sup>A</sup>T<sub>E</sub>X on a file containing just the picture.

Make a coordinate grid with the `\graphpaper` command, and use it for placing objects.

If you are not afraid of math, you will find that a few simple calculations can save a lot of trial and error—especially when drawing slanted lines and arrows.

It's a good idea to break a complicated picture into “subpictures”. The subpicture is drawn in a separate `picture` environment inside a `\put` argument, as in

```
\put(13,14.2){\begin{picture}(10,7) ... \end{picture}}
```

This permits easy repositioning of the subpicture and allows you to work in terms of local coordinates relative to the subpicture's origin instead of calculating the position of every picture component with respect to a single origin. You can also magnify or reduce just the subpicture by changing the value of `\unitlength` with a `\setlength` command in the `\put` command's argument—but don't leave any space after the `\setlength` command.

A small mistake in a picture-drawing command can produce strange results. It's usually simple to track down such an error, so don't panic when a picture turns out all wrong. If you find that some part of the picture is incorrectly positioned by a small amount, check for stray spaces in the argument of the `\put` command. Remember that this argument is typeset in LR mode, so a space before or after an object produces space in the output.

## 7.2 The graphics Package

The `graphics` package defines commands for performing geometric transformations and including graphics prepared with other computer programs. The geometric transformations are scaling, rotation, and reflection. These transformations may be applied to any text including ordinary letters and words, but they are usually applied to pictures made with the `picture` environment and included graphics. This package requires support from the device driver; see the remarks at the beginning of this chapter.

The package provides two scaling commands, `\scalebox` and `\resizebox`. The `\scalebox` command allows you to enlarge or reduce any text by a constant scale factor.

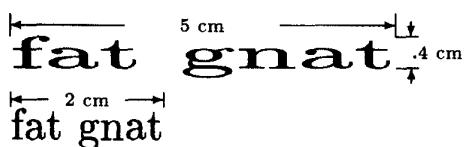
great big gnat  
regular gnat  
tiny gnat

```
\scalebox{2}{great big gnat}
regular gnat
\scalebox{.5}{tiny gnat}
```

An optional second argument specifies a separate vertical scale factor.

tall skinny gnat `\scalebox{.75}[2]{tall skinny gnat}`

The `\resizebox` allows you to scale text to a desired size. Its two arguments specify the width and height of the box that it produces. Using `!` for either argument maintains the aspect ratio of the text.

  
fat gnat  
fat gnat

```
\resizebox{5cm}{.4cm}{fat gnat}
\resizebox{2cm}{!}{fat gnat}
```

Both `\scalebox` and `\resizebox` typeset their text argument in LR mode. They produce a box whose reference point is the same as the reference point of the original unscaled box.

The `\rotatebox` command rotates text by a specified angle. In the following examples, the sizes and reference points of boxes are shown.



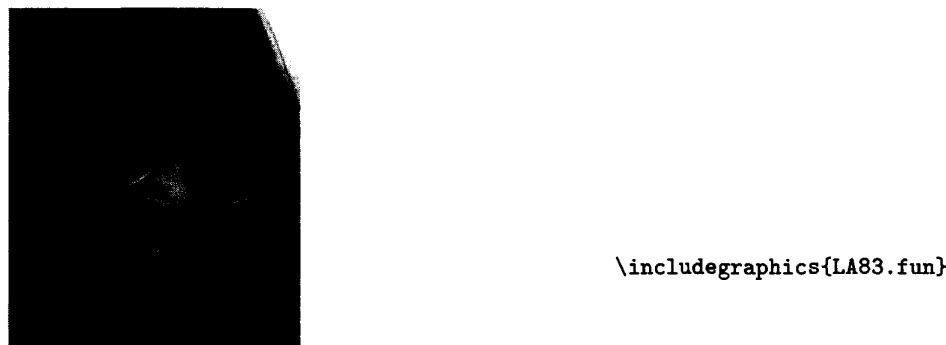
The `\rotatebox` command typesets its text argument in LR mode. It produces a box whose reference point is at the same height as the rotated reference point of the original text.



The `\reflectbox` command typesets its argument in LR mode and then produces its mirror image.

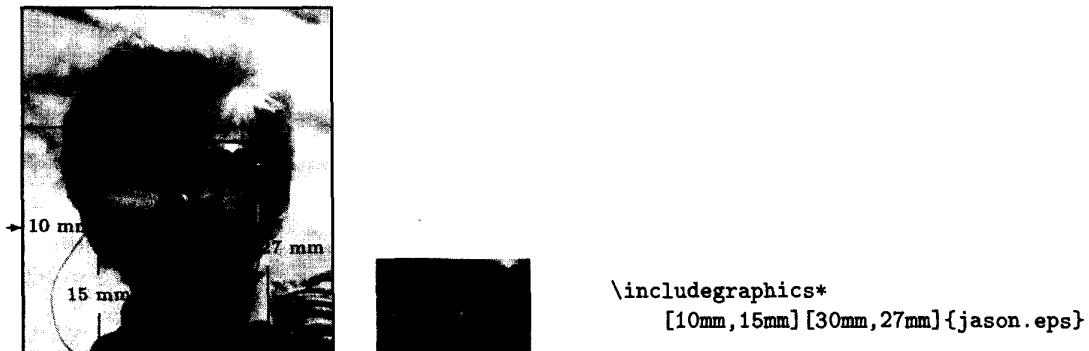


Graphics produced by another program can be included in your document with the `\includegraphics` command. The following example assumes that the file `LA83.fun` contains a picture in a form that the device driver can handle.



In this example, the size of the box is specified by the file `LA83.fun`. Some graphics files do not include a size specification. In that case, L<sup>A</sup>T<sub>E</sub>X will produce an error message, and you will have to specify the size yourself using optional arguments to the `\includegraphics` command. In the next example, the first

picture shows what is actually in the file `jason.eps` and the second shows the output of the `\includegraphics` command.<sup>2</sup>



The optional arguments to `\includegraphics` override any size specification in the file itself. However, the box specified by the file (and produced by `\includegraphics` with no optional argument) might not contain the entire picture. A command such as

```
\resizebox{\textwidth}{!}
{\includegraphics[0in,0in][8in,10in]{jason.eps}}
```

will show you what is actually in the file `jason.eps`.

The `draft` option causes the `\includegraphics` command to print a box containing the name of the file rather than the contents of the file. This option is specified either by typing

```
\usepackage[draft]{graphics}
```

or by specifying the `draft` document-class option (Section 2.2.2). A `final` option in the `\usepackage` command counteracts the effects of a `draft` document-class option.

## 7.3 Color

You can produce colored text with the `\textcolor` command or the corresponding `\color` declaration. (To keep the cost of this book down, colors are indicated by shades of gray.)

CECI N'EST PAS UNE PIPE ROUGE.

... UNE \textcolor{red}{PIPE} ROUGE.

CECI N'EST PAS UNE PIPE NOIRE.

\color{red}
 ... UNE {\color{black} PIPE} NOIRE.

<sup>2</sup>Without the `*` after `\includegraphics`, the entire picture would be printed, although TeX would leave only enough space for the small rectangle.

The `\colorbox` command typesets its argument in LR mode on a rectangular background of a specified color.



```
\colorbox{red}{UNE PIPE ROUGE}
```

The command `\fcolorbox{blue}{red}{UNE...}` puts a blue border around the red rectangle.

A `\pagecolor{green}` declaration causes the background of the entire page to be green. It is a global declaration that applies to the current page and all subsequent pages, until countermanded by another `\pagecolor` command.

The following colors are predefined: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, and `magenta`. The `\definecolor` command allows you to define your own colors in terms of a *color model*. In the `rgb` color model, a color is specified by three numbers, ranging from 0 to 1, that represent the amounts of red, green, and blue light required to produce it. For example, magenta is produced by mixing equal amounts of red and blue light, so it is defined by

```
\definecolor{magenta}{rgb}{1,0,1}
```

A darker shade of magenta is defined by

```
\definecolor{darkmagenta}{rgb}{.5,0,.5}
```

In the `gray` color model, a shade of gray is specified by a single number ranging from 0 (black) to 1 (white).



```
\definecolor{dark}{gray}{.5}
\colorbox{dark}{A dark gray box.}
```



```
\definecolor{light}{gray}{.75}
\colorbox{light}{A light gray box.}
```

`LATEX` also supports the `cmyk` model, popular with (human) printers, in which a color is specified by values of cyan, magenta, yellow, and black ranging from 0 to 1. Other color models may be available on your computer, including ones in which a color is specified by a name. Check your *Local Guide*.

After `LATEX 2E` is released, there may be a period when some drivers will not handle color commands properly. Check your output carefully, especially when the scope of a color declaration spans pages—for example if the declaration appears in a footnote that is split across pages.

# CHAPTER 8

# Errors



Section 2.3 describes first aid for handling errors; it explains how to deal with simple errors. This chapter is for use when you encounter an error or **warning** message that you don't understand. The following section tells how to locate the error; the remaining sections explain the meaning of specific error and warning messages.

As you saw in Section 2.3, an error can confuse L<sup>A</sup>T<sub>E</sub>X and cause it to produce spurious error messages when processing subsequent text that is perfectly all right. Such spurious errors are not discussed here. When T<sub>E</sub>X writes a page of output, it has usually recovered from the effects of any previous errors, so the next error message probably indicates a real error. The following section explains how to tell when T<sub>E</sub>X has written an output page.

## 8.1 Finding the Error

As described in Section 2.3, an error message includes an error indicator stating what T<sub>E</sub>X thinks the problem is, and an error locator that shows how much of your input file T<sub>E</sub>X had read when it found the error. Most of the time, the line printed in the error locator displays an obvious error in the input. Typing H can also provide useful information. If you don't see what's wrong, look up the error message in the following sections to find its probable cause. If you still don't see what's wrong, the first thing to do is locate exactly where the error occurred.

The error locator starts with a line number such as 1.14, meaning that the error was found while L<sup>A</sup>T<sub>E</sub>X was processing the fourteenth line from the beginning of the file. If your input is all on a single file, then the error locator unambiguously identifies where T<sub>E</sub>X thinks the problem is. However, if you're using the commands of Section 4.4 to split your input into several files, then you also must know what file the error is in. Whenever T<sub>E</sub>X starts processing a file, it prints on your terminal (and on the log file) a "(" followed by the file name, and it prints a ")" when it finishes processing the file. For example, the terminal output

```
... (myfile.tex [1] [2] [3] (part1.tex [4] [5]) (part2.tex [6] [7]
! Undefined control sequence.
1.249 \todzy
?
```

tells you that the error (a misspelled \today command) is on line 249 of the file *part2.tex*, which was included by an \input or \include command contained in the file *myfile.tex*. T<sub>E</sub>X had completely processed the file *part1.tex*, which was also read by a command in *myfile.tex*.

The error locator tells you how much of the input file T<sub>E</sub>X had processed before it discovered the error; the last command that T<sub>E</sub>X processed is usually the source of the problem. There is one important exception; but before discussing it, a digression is in order.

Logically, typesetting can be viewed as a two-step process: first the document is typeset on one continuous scroll that unrolls vertically, then the scroll is cut into individual pages to which headings and page numbers are added. (Since a 50-yard scroll of metal type is somewhat unwieldy, printers partition the logical scroll into convenient lengths called galleys.) Instead of first producing the entire scroll and then cutting it into pages,  $\text{\TeX}$  does both steps together, alternately putting output on the scroll with one hand and cutting off a page with the other. It usually puts text on the scroll one paragraph unit (Section 6.2.1) at a time. After each paragraph unit, it checks whether there's enough for a page. If so, it cuts off the page, adds the heading and page number, and writes the page on the *dvi* file. This way,  $\text{\TeX}$  doesn't have to keep much more than one page of text in the computer's memory at a time.

Whenever  $\text{\TeX}$  writes a page on its output file, it prints the page number on the terminal, enclosed in square brackets. Thus, any message that appears on the terminal after  $\text{\TeX}$  prints [27] and before it prints [28] was generated between the time  $\text{\TeX}$  wrote output pages 27 and 28. Whatever generated the message probably appeared in the text printed on page 28. However, it might also be in the text that was left on the scroll when  $\text{\TeX}$  cut off page 28, putting it in the first paragraph of page 29. Most of  $\text{\LaTeX}$ 's warning messages are generated by  $\text{\TeX}$ 's scroll-making hand. It reports that a problem is on page 28 if it is detected between the time  $\text{\TeX}$  writes pages 27 and 28, so the problem could actually appear at the top of page 29.

Now, let's get back to locating an error. Most errors are discovered while  $\text{\TeX}$  is producing the scroll, but some errors, which I will call *outputting* errors, are detected while it is cutting off a page.  $\text{\TeX}$  identifies an outputting error by printing `<output>` on the terminal at the beginning of a line somewhere above the error locator. For an outputting error, the error locator shows how far  $\text{\TeX}$  got when it was producing the scroll; the actual error occurred at or before that point. An outputting error is usually caused by a fragile command in a moving argument.

An error can occur while  $\text{\LaTeX}$  is processing the `\begin{document}` or `\end{document}` command, when it reads the auxiliary files it has written. You can tell that this has happened because the output on your terminal shows that  $\text{\TeX}$  is reading a file with the extension `aux`. Such an error means that there is bad information on the auxiliary file. An error while processing the `\begin{document}` was caused by an error the last time you ran  $\text{\LaTeX}$ . An error while processing the `\end{document}` is usually caused by a fragile command in a moving argument.

An error that occurs while  $\text{\LaTeX}$  is processing a `toc`, `lof`, or `lot` file means that there is an error in a table of contents, list of figures, or list of tables entry. This error was caused by an error in a `\caption` or sectioning command—perhaps a fragile command in the argument—the previous time  $\text{\LaTeX}$  was run on the file.

When the terminal output doesn't quickly lead you to the source of the error, look at the printed output (on your screen previewer). If L<sup>A</sup>T<sub>E</sub>X reaches the end of your input or is stopped with a `\stop` command, the printed output will contain everything it has put on the scroll, and the location of the error will probably be obvious. If you stopped L<sup>A</sup>T<sub>E</sub>X by typing an `X`, then it will not print what was left on the scroll after the last full page was written out. Since the error probably occurred in this leftover text, the output will just narrow the possible location of the error.

If you still can't find the error, your next step is to find the smallest piece of your input file that produces the error. Start by eliminating everything between the `\begin{document}` and the last page or so of output. Then keep cutting the input in half, throwing away the part that does not cause the error. This should quickly lead to the source of the problem.

When all else fails, consult your *Local Guide* to find a L<sup>A</sup>T<sub>E</sub>X expert near you.

## 8.2 L<sup>A</sup>T<sub>E</sub>X's Error Messages

Here is an alphabetical list of most of L<sup>A</sup>T<sub>E</sub>X's error indicators (with the initial “! L<sup>A</sup>T<sub>E</sub>X Error:” omitted), together with their causes. Not listed are indicators for errors caused by a bad document class, some errors for which typing `H` provides a clear explanation, and many errors produced only when a particular package is used.

### **Bad \line or \vector argument.**

A `\line` or `\vector` command specified a negative length or an illegal slope. Look up the constraints on these commands in Section 7.1.

### **Bad math environment delimiter.**

L<sup>A</sup>T<sub>E</sub>X has found either a math-mode-starting command such as `\[` or `\(` when it is already in math mode, or else a math-mode-ending command such as `\]` or `\)` while in LR or paragraph mode. The problem is caused by either unmatched math mode delimiters or unbalanced braces.

### **\begin{...} on input line ... ended by \end{...}.**

L<sup>A</sup>T<sub>E</sub>X has found an `\end` command that doesn't match the corresponding `\begin` command. You probably misspelled the environment name in the `\end` command, have an extra `\begin`, or else forgot an `\end`.

### **Can be used only in preamble.**

L<sup>A</sup>T<sub>E</sub>X has encountered, after the `\begin{document}`, a command that should appear only in the preamble, such as `\usepackage`, `\nofiles`, `\includeonly`, or `\makeindex`. The error is also caused by an extra `\begin{document}` command.

**Cannot determine size of graphic in ... (no BoundingBox).**

An `\includegraphics` command with no optional arguments read a file that does not specify the size of the box to be produced (`graphics` package only).

**Command ... already defined.**

You are using `\newcommand`, `\newenvironment`, `\newlength`, `\newsavebox`, or `\newtheorem` to define a command or environment name that is already defined, or `\newcounter` to define a counter that already exists. (Defining an environment named `gnu` automatically defines the command `\gnu`.) You'll have to choose a new name or, in the case of `\newcommand` or `\newenvironment`, switch to the `\renew...` command.

**Command ... invalid in math mode.**

The indicated command is not permitted in math mode but was used there.

**Counter too large.**

Some object that is numbered with letters or with footnote symbols has received too large a number. You're probably either making a very long enumerated list or resetting counter values.

**Environment ... undefined.**

L<sup>A</sup>T<sub>E</sub>X has encountered a `\begin` command for a nonexistent environment. You probably misspelled the environment name. This error can be corrected on the spot by typing an `I` followed by the correct command, ending with a *return*. (This does not change the input file.)

**File ... not found.**

L<sup>A</sup>T<sub>E</sub>X is trying to read a file that doesn't exist. If the missing file has the extension `tex`, then it is trying to `\input` or `\include` a nonexistent file; if it has the extension `cls`, then you have specified a nonexistent document class; if it has the extension `sty`, then you have specified a nonexistent package. L<sup>A</sup>T<sub>E</sub>X is waiting for you either to type another file name followed by *return*, or to type *return* to continue without reading any file.

**Illegal character in array arg.**

There is an illegal character in the argument of an `array` or `tabular` environment, or in the second argument of a `\multicolumn` command.

**\include cannot be nested.**

Your document `\include`'s a file containing an `\include` command.

**Lonely \item--perhaps a missing list environment.**

An `\item` command appears outside any list environment.

**Missing `\begin{document}`.**

`LATEX` produced printed output before encountering a `\begin{document}` command. Either you forgot the `\begin{document}` command or there is something wrong in the preamble. The problem may be a stray character or an error in a declaration—for example, omitting the braces around an argument or forgetting the `\` in a command name.

**Missing `p`-arg in array arg.**

There is a `p` that is not followed by an expression in braces in the argument of an `array` or `tabular` environment, or in the second argument of a `\multicolumn` command.

**Missing `Q`-exp in array arg.**

There is an `Q` character not followed by an `Q`-expression in the argument of an `array` or `tabular` environment, or in the second argument of a `\multicolumn` command.

**No counter ‘...’ defined.**

You have specified a nonexistent counter in a `\setcounter` or `\addtocounter` command, or in an optional argument to a `\newcounter` or `\newtheorem` command. You probably mistyped the counter name. However, if the error occurred while a file with the extension `aux` is being read, then you probably used a `\newcounter` command in an `\include`'d file.

**No `\title` given.**

A `\maketitle` command is not preceded by a `\title` command.

**Not in outer par mode.**

You had a `figure` or `table` environment or a `\marginpar` command in math mode or inside a `parbox`.

**Option clash for package ...**

The same package was loaded twice with different options. The package might have been loaded by another package.

**`\pushtabs` and `\poptabs` don't match.**

`LATEX` found a `\poptabs` with no matching `\pushtabs`, or has come to the `\end{tabbing}` command with one or more unmatched `\pushtabs` commands.

**Something's wrong--perhaps a missing `\item`.**

There are many possible causes, including an omitted `\item` command in a list-making environment and a missing argument of a `\bibliography` environment.

**Tab overflow.**

A `\=` command has exceeded the maximum number of tab stops that *L<sup>A</sup>T<sub>E</sub>X* permits.

**There's no line here to end.**

A `\newline` or `\\"` command appears between paragraphs, where it makes no sense. If you are trying to “leave a blank line”, use a `\vspace` command (Section 6.4.2).

**This file needs format ... but this is ...**

The document uses a document class or package that is not compatible with the version of *L<sup>A</sup>T<sub>E</sub>X* you are running. If you are using only standard files, then there is something wrong with the installation of *L<sup>A</sup>T<sub>E</sub>X* on your computer.

**This may be a *LaTeX* bug.**

*L<sup>A</sup>T<sub>E</sub>X* has become thoroughly confused. This is probably due to a previously detected error, but it is possible that you have discovered an error in *L<sup>A</sup>T<sub>E</sub>X* itself. If this is the first error message produced by the input file and you can't find anything wrong, save the file and contact the person listed in your *Local Guide*.

**Too deeply nested.**

There are too many list-making environments nested within one another. How many levels of nesting are permitted may depend upon what computer you are using, but at least four levels are provided, which should be enough.

**Too many columns in *eqnarray* environment.**

An *eqnarray* environment contains three & column separators without an intervening `\\"` command.

**Too many unprocessed floats.**

While this error can result from having too many `\marginpar` commands on a page, a more likely cause is forcing *L<sup>A</sup>T<sub>E</sub>X* to save more figures and tables than it has room for. When typesetting its continuous scroll, *L<sup>A</sup>T<sub>E</sub>X* saves figures and tables separately and inserts them as it cuts off pages. This error occurs when *L<sup>A</sup>T<sub>E</sub>X* is forced to save too many `figure` and/or `table` environments. A likely cause is a logjam—a figure or table that cannot be printed, causing others to pile up behind it, since *L<sup>A</sup>T<sub>E</sub>X* will not print figures or tables out of order. The jam can be started by a figure or table that won't fit where its optional placement argument (Section C.9.1) says it must go. See the discussion of *L<sup>A</sup>T<sub>E</sub>X*'s figure-placement algorithm in Section C.9.1.

**Undefined color ‘...’.**

The indicated color name is used without having been defined by a `\definecolor` command (`color` package only).

**Undefined color model ‘...’.**

A `\definecolor` command specifies an unknown color model (`color` package only).

**Undefined tab position.**

A `\>`, `\+`, `\-`, or `\<` command is trying to go to a nonexistent tab position—one not defined by a `\=` command.

**Unknown graphics extension ...**

The `graphics` package's `\includegraphics` command uses the file's extension to determine what kind of program produced the file. This error occurs when the file name does not have an extension known to the package. Consult the *Local Guide* to see what file extensions you can use.

**Unknown option ... for ...**

The `\documentclass` command or a `\usepackage` command specifies an illegal option for the class or package.

**\verb ended by end of line.**

The argument of a `\verb` command extends beyond the current line. You may have forgotten the character that ends the argument.

**\verb illegal in command argument.**

A `\verb` command appears in the argument of another command.

**\< in mid line.**

A `\<` command appears in the middle of a line in a `tabbing` environment. This command should come only at the beginning of a line.

### 8.3 T<sub>E</sub>X's Error Messages

Here is an alphabetical list of some of T<sub>E</sub>X's error messages and what may have caused them.

**! Double subscript.**

There are two subscripts in a row in a mathematical formula—something like `x_{2}_{3}`, which makes no sense. To produce  $x_{2_3}$ , type `x_{2_{3}}`.

**! Double superscript.**

There are two superscripts in a row in a mathematical formula—something like  $x^{2^3}$ , which makes no sense. To produce  $x^{2^3}$ , type  $x^{2^{3}}$ .

**! Extra alignment tab has been changed to \cr.**

There are too many separate items (column entries) in a single row of an **array** or **tabular** environment. In other words, there were too many **&**'s before the end of the row. You probably forgot the **\backslash** at the end of the preceding row.

**! Extra }, or forgotten \$.**

The braces or math mode delimiters don't match properly. You probably forgot a **{**, **\[**, **\(**, or **\$**.

**! I can't find file '...`.**

You probably ran **L<sup>A</sup>T<sub>E</sub>X** on a nonexistent file. This error can also occur if you omitted the braces around an **\input** command.

**! Illegal parameter number in definition of ... .**

This is probably caused by a **\newcommand**, **\renewcommand**, **\providecommand**, **\newenvironment**, or **\renewenvironment** command in which a **#** is used incorrectly. A **#** character, except as part of the command name **\#**, can be used only to indicate an argument parameter, as in **#2**, which denotes the second argument. This error is also caused by nesting one of the five commands listed above inside another, or by putting a parameter like **#2** in the last argument of a **\newenvironment** or **\renewenvironment** command.

**! Illegal unit of measure (pt inserted).**

If you just got a

**! Missing number, treated as zero.**

error, then this is part of the same problem. If not, it means that **L<sup>A</sup>T<sub>E</sub>X** was expecting a length as an argument and found a number instead. The most common cause of this error is writing **0** instead of something like **0in** for a length of zero, in which case typing **return** should result in correct output. However, the error can also be caused by omitting a command argument.

**! Misplaced alignment tab character &.**

The special character **&**, which should be used only to separate items in an **array** or **tabular** environment, appeared in ordinary text. You probably meant to type **\&**, in which case typing **I\&** followed by **return** in response to the error message should produce the correct output.

**! Missing control sequence inserted.**

This is probably caused by a `\newcommand`, `\renewcommand`, `\newlength`, or `\newsavebox` command whose first argument is not a command name.

**! Missing number, treated as zero.**

This is usually caused by a L<sup>A</sup>T<sub>E</sub>X command expecting but not finding either a number or a length as an argument. You may have omitted an argument, or a square bracket in the text may have been mistaken for the beginning of an optional argument. This error is also caused by putting `\protect` in front of either a length command or a command such as `\value` that produces a number.

**! Missing { inserted.**

**! Missing } inserted.**

T<sub>E</sub>X has become confused. The position indicated by the error locator is probably beyond the point where the incorrect input is.

**! Missing \$ inserted.**

T<sub>E</sub>X probably found a command that can be used only in math mode when it wasn't in math mode. Remember that unless stated otherwise, all the commands of Section 3.3 can be used only in math mode. T<sub>E</sub>X is not in math mode when it begins processing the argument of a box-making command, even if that command is inside a math environment. This error also occurs if T<sub>E</sub>X encounters a blank line when it is in math mode.

**! Not a letter.**

Something appears in the argument of a `\hyphenation` command that doesn't belong there.

**! Paragraph ended before ... was complete.**

A blank line occurred in a command argument that shouldn't contain one. You probably forgot the right brace at the end of an argument.

**! TeX capacity exceeded, sorry [...].**

T<sub>E</sub>X has just run out of space and aborted its execution. Before you panic, remember that the least likely cause of this error is T<sub>E</sub>X not having the capacity to process your document. It was probably an error in your input file that caused T<sub>E</sub>X to run out of room. For example, the following command makes a circular definition, defining `\gnu` in terms of itself:

```
\newcommand{\gnu}{a \gnu} % This is wrong!
```

When T<sub>E</sub>X encounters this `\gnu` command, it will keep chasing its tail trying to figure out what `\gnu` should produce, and eventually run out of space. L<sup>A</sup>T<sub>E</sub>X seldom runs out of space on a short input file, so if running it on the last few

pages before the error indicator's position still produces the error, then there's almost certainly something wrong in the input file. An error that causes **TeX** to run out of space can be hard to find; you may have to use the divide and conquer method described in Section 8.1 to locate it.

Today's computers have enough memory to provide **TeX** with all the space it needs to process almost any document. However, any particular version of **TeX** uses only a fixed amount of space, and the version on your computer may not be big enough to process your document. In this case, you should try to get a bigger version. Meanwhile, by understanding why **TeX** ran out of space, you can reduce the space it needs for your document. The end of the error indicator tells what kind of space **TeX** ran out of. The kinds of space you are most likely to run out of are listed below, with an explanation of what uses them up.

**buffer size** Can be caused by too long a piece of text as the argument of a sectioning, `\caption`, `\addcontentsline`, or `\addtocontents` command. This error will probably occur when the `\end{document}` is being processed, but it could happen when a `\tableofcontents`, `\listoffigures`, or `\listoftables` command is executed. To solve this problem, use a shorter optional argument. Even if you're producing a table of contents or a list of figures or tables, such a long entry won't help the reader.

**exception dictionary** You have used `\hyphenation` commands to give **TeX** more hyphenation information than it has room for. Remove some of the less frequently used words from the `\hyphenation` commands and insert `\-` commands instead.

**hash size** Your input file defines too many command names and/or uses too many cross-referencing labels.

**main memory size** This is one kind of space that **TeX** can run out of when processing a short file. There are three ways you can run **TeX** out of main memory space: (1) defining a lot of very long, complicated commands, (2) making an index or glossary and having too many `\index` or `\glossary` commands on a single page, and (3) creating so complicated a page of output that **TeX** can't hold all the information needed to generate it.

The solution to the first two problems is obvious: define fewer commands or use fewer `\index` and `\glossary` commands. The third problem is nastier. It can be caused by large `tabbing`, `tabular`, `array`, and `picture` environments. (In particular, `\qbezier` commands can use a lot of main memory.) **TeX**'s space may also be filled up with figures and tables waiting for a place to go.

To find out if you've really exceeded **TeX**'s capacity in this way, put a `\clearpage` command in your input file right before the place where **TeX** ran out of room and try running it again. If it doesn't run out of room

with the `\clearpage` command there, then you did exceed `TeX`'s capacity. If it still runs out of room, then there's probably an error in your file.

If `TeX` is really out of room, you must give it some help. Remember that `TeX` processes a complete paragraph before deciding whether to cut the page. Inserting a `\newpage` command in the middle of the paragraph, where `TeX` should break the page, may save the day by letting `TeX` write out the current page before processing the rest of the paragraph. (A `\pagebreak` command won't help.) If the problem is caused by accumulated figures and tables, you can try to prevent them from accumulating—either by moving them further towards the end of the document or by trying to get them to come out sooner. (See Section C.9.1 for more details.) If you are still writing the document, simply add a `\clearpage` command and forget about the problem until you're ready to produce the final version. Changes to the input file are likely to make the problem go away.

**pool size** You probably used too many cross-referencing labels and/or defined too many new command names. More precisely, the labels and command names that you define have too many characters, so this problem can be solved by using shorter names. However, the error can also be caused by omitting the right brace that ends the argument of either a counter command such as `\setcounter`, or a `\newenvironment` or `\newtheorem` command.

**save size** This occurs when commands, environments, and the scopes of declarations are nested too deeply—for example, by having the argument of a `\multiput` command contain a `picture` environment that in turn has a `\footnotesize` declaration whose scope contains a `\multiput` command containing a ...

**! Text line contains an invalid character.**

The input file contains some strange nonprinting character (sometimes called a control character) that it shouldn't. Instead of using a simple text editor, you may have created the file with a program that does its own formatting. Such programs usually insert strange characters into a file unless you save it as a text (or ASCII) file.

**! Undefined control sequence.**

`TeX` encountered an unknown command name. You probably misspelled the name, in which case typing I followed by the desired command name and a `return` will produce correct output. However, you still must change the input file later. If this message occurs when a `LATEX` command is being processed, the command is probably in the wrong place. The error can also be caused by a missing `\documentclass` or `\usepackage` command.

**! Use of ... doesn't match its definition.**

If the “...” is a L<sup>A</sup>T<sub>E</sub>X command, then it's probably one of the picture-drawing commands described in Section 7.1, and you have used the wrong syntax for specifying an argument. If it's `\array` that doesn't match its definition, then there is something wrong in an `\@-expression` in the argument of an `array` or `tabular` environment—perhaps a fragile command that is not `\protect`'ed. The error can also be caused by an un`\protect`'ed command with an optional argument appearing in a moving argument. (The “...” may or may not be the name of the command.)

**! You can't use 'macro parameter character #' in ... mode.**

The special character `#` has appeared in ordinary text. You probably meant to type `\#`, in which case you can respond to the error message by typing `I\#` followed by *return* to produce the correct output.

## 8.4 L<sup>A</sup>T<sub>E</sub>X Warnings

L<sup>A</sup>T<sub>E</sub>X's warning messages begin with “L<sup>A</sup>T<sub>E</sub>X Warning:”. (Some begin with an indication of the class of warning—for example, “L<sup>A</sup>T<sub>E</sub>X Font Warning:”.) The most common messages are described below. Certain classes of warnings cause L<sup>A</sup>T<sub>E</sub>X to print an additional message at the end to indicate that a warning of that class occurred.

**Citation '...' on page ... undefined.**

The citation key in a `\cite` command was not defined by a `\bibitem` command. See Section 4.3.

**Command ... invalid in math mode.**

The indicated command is not permitted in math mode but was used there. Remember that `\boldmath`, `\unboldmath`, and size-changing commands may not be used in math mode.

**Float too large for page by ...**

A figure or table is too tall, by the indicated length, to fit on a page. It is printed by itself on an oversize page. The length is given in points.

**Font shape '...' in size ... not available.**

You specified a font that is not available on your computer. The next line of the message describes what font L<sup>A</sup>T<sub>E</sub>X is using in its place. As explained in Section 3.3.8, this message can be caused by a `\boldmath` declaration even if the font is never used.

**h float specifier changed to ht**  
**!h float specifier changed to !ht**

A figure or table environment has an optional argument `h` or `!h`, but the figure or table would not fit on the current page, so it is being put on the top of a subsequent page.

**Label '...' multiply defined.**

Two `\label` or `\bibitem` commands have the same arguments. More precisely, they had the same arguments the preceding time that L<sup>A</sup>T<sub>E</sub>X processed the input.

**Label(s) may have changed. Rerun to get cross-references right.**

Issued after processing the entire input if the numbers printed by `\cite`, `\ref`, or `\pageref` commands may be wrong because the correct values have changed since the last time L<sup>A</sup>T<sub>E</sub>X processed the input.

**Marginpar on page ... moved.**

A marginal note was moved down on the page to avoid printing it on top of a previous marginal note. It will therefore not be aligned with the line of text where the `\marginpar` command appeared.

**No \author given.**

A `\maketitle` command is not preceded by an `\author` command.

**Optional argument of \twocolumn too tall on page ...**

The optional argument of a `\twocolumn` command produced a box too large to fit on a page.

**Oval too small.**

An `\oval` command specified an oval so small that L<sup>A</sup>T<sub>E</sub>X could not draw small enough quarter-circles to put in its corners. What L<sup>A</sup>T<sub>E</sub>X did draw does not look very good.

**Reference '...' on page ... undefined.**

The argument of a `\ref` or `\pageref` command was not defined by a `\label` command. See Section 4.2.

**Some font shapes were not available, defaults substituted.**

Issued after processing the entire input if any unavailable font was specified.

**There were multiply-defined labels.**

Issued after processing the entire input if any label was defined by two different `\label` commands.

**There were undefined references or citations.**

Issued after processing the entire input if a `\ref` or `\cite` was found that had no corresponding `\label` or bibliography entry.

**Unused global option(s): [...] .**

The listed options were given to the `\documentclass` command but were not known to it or to any packages that were loaded.

**You have requested release ‘...’ of *LaTeX*, but only release  
‘...’ is available.**

You are using a document class or package that requires a later release of *LaTeX* than the one you are running. You should get the latest release of *LaTeX*.

## 8.5 **T<sub>E</sub>X Warnings**

You can identify a *T<sub>E</sub>X* warning message because it is not an error message, so no ? is printed, and it does not begin with “*LaTeX Warning:*”. Below is a list of some *T<sub>E</sub>X*’s warnings.

**Overfull \hbox ...**

See Section 6.2.1.

**Overfull \vbox ...**

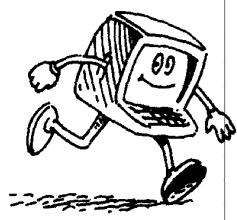
Because it couldn’t find a good place for a page break, *T<sub>E</sub>X* put more on the page than it should. See Section 6.2.2 for how to deal with page-breaking problems.

**Underfull \hbox ...**

Check your output for extra vertical space. If you find some, it was probably caused by a problem with a \\ or `\newline` command—for example, two \\ commands in succession. This warning can also be caused by using the `sloppypar` environment or `\sloppy` declaration, or by inserting a `\linebreak` command.

**Underfull \vbox ...**

*T<sub>E</sub>X* could not find a good place to break the page, so it produced a page without enough text on it. See Section 6.2.2 for how to handle page-breaking problems.



## APPENDIX A

# Using *MakeIndex*



## A.1 How to Use *MakeIndex*

*MakeIndex* is a program for making an index from information generated by `\index` commands in your document. Section A.2 below explains what `\index` commands to use to produce the index entries you want. To use *MakeIndex*, you must also put the following commands in your document:

- `\usepackage{makeidx}` in the preamble (between the `\documentclass` and `\begin{document}` commands).
- `\makeindex` in the preamble.
- `\printindex` where you want the index to appear—usually right before the `\end{document}` command.

Let's suppose that the root file (Section 4.4) of your document is `myfile.tex`. You first run *L<sup>A</sup>T<sub>E</sub>X* on your entire document, which causes it to generate the file `myfile.idx` containing the information from your `\index` commands. You next run *MakeIndex* by typing something like

```
makeindex myfile
```

*MakeIndex* reads `myfile.idx` and produces the file `myfile.ind`. *MakeIndex* will reject an entry or issue a warning when it finds an error. If it doesn't find any, you can rerun *L<sup>A</sup>T<sub>E</sub>X* on your document; *L<sup>A</sup>T<sub>E</sub>X* will use `myfile.ind` to produce the index. If *MakeIndex* does find errors, see Section A.3 below.

By reading the index, you may discover additional mistakes. These should be corrected by changing the appropriate `\index` commands in the document and regenerating the `ind` file. If there are problems that cannot be corrected in this way, you can always edit the `ind` file directly. However, such editing is to be avoided because it must be repeated every time you generate a new version of the index.

*MakeIndex* can be customized in a variety of ways to make glossaries and other kinds of indexes. See the *L<sup>A</sup>T<sub>E</sub>X Companion* for details.

## A.2 How to Generate Index Entries

### A.2.1 When, Why, What, and How to Index

The index is there to help the reader find things in your document. It should make this as easy as possible. Many authors index words, listing all the pages on which a word appears. A good writer indexes concepts—ideas, facts, people, etc. Here is an entry from a word index and the corresponding entry from a concept index. Imagine using each of them to find out if there are gnus in Tasmania. (Page 150 tells you that there aren't.)

gnu, 17, 25, 54, 62, 64, 74, 101,  
103, 104, 121, 124, 125,  
150, 167, 202, 250

gnu,  
caged, 104, 121, 125  
distribution of, 25, 54, 150, 167  
gnat and, 62, 64, 103, 124, 202  
indexing, 74, 150, 250  
size of, 17, 25, 101, 167

With the word index, you have to look at twelve pages before reaching the right one; with the concept index, you have to look at only two.

To make an index, you must first decide what concepts should appear in it. You must then figure out under what words a reader might look to find each concept. Try to understand who your readers are and how they think about the concept. Don't just list the words that you used to describe it.

You may be tempted to generate the index as you write the document. Resist the temptation. It is virtually impossible to make a good index that way. Add `\index` commands as you write to remind yourself of what you want to index, but be prepared to modify those commands when you produce the index.

A computer can't generate the index for you, but it can help. You have to decide whether gnus are central enough to your topic to appear in the index, and if so, where to direct the reader who wants to learn about them. The computer can help by finding all occurrences of "gnu". A good way to start writing an index is by making an alphabetized list of all the distinct words that appear in your document. Consult your *Local Guide* to see if your computer has a program to generate such a list.

Most books have indexes; most technical reports don't. They should. Any nonfiction work of more than twenty or so pages that is worth reading deserves an index. With L<sup>A</sup>T<sub>E</sub>X and *MakeIndex* doing the tedious work for you, there is no good reason not to make one.

### A.2.2 The Basics

The following example shows some simple `\index` commands and the index entries that they produce using *MakeIndex*. The page number refers to the page containing the text where the `\index` command appears.

|                     |                            |
|---------------------|----------------------------|
| Alpha, ii           | Page ii: \index{Alpha}     |
| alpha, viii, ix, 22 | Page viii: \index{alpha}   |
| alpha bet, 24       | Page ix: \index{alpha}     |
| Alphabet, ix        | \index{Alphabet}           |
| alphabet, 23        | Page 22: \index{alpha}     |
| alphas, 22          | \index{alphas}             |
|                     | Page 23: \index{alphabet}  |
|                     | \index{alphabet}           |
|                     | Page 24: \index{alpha bet} |

The duplicate `\index{alphabet}` commands on page 23 produce only one “23” in the index.

To produce a subentry, the argument of the `\index` command should contain both the main entry and the subentry, separated by a ! character.

|             |                                            |
|-------------|--------------------------------------------|
| gnat, 32    | Page 7: <code>\index{gnat!size of}</code>  |
| anatomy, 35 | Page 32: <code>\index{gnat}</code>         |
| size of, 7  | Page 35: <code>\index{gnat!anatomy}</code> |
| gnus        | <code>\index{gnus!good}</code>             |
| bad, 38     | Page 38: <code>\index{gnus!bad}</code>     |
| good, 35    |                                            |

You can also have subsubentries.

|               |                                                 |
|---------------|-------------------------------------------------|
| bites         | Page 8: <code>\index{bites!animal!gnats}</code> |
| animal        | Page 10: <code>\index{bites!animal!gnus}</code> |
| gnats, 8      | Page 12: <code>\index{bites!vegetable}</code>   |
| gnus, 10      |                                                 |
| vegetable, 12 |                                                 |

*L<sup>A</sup>T<sub>E</sub>X* and *MakeIndex* support only three levels of indexing; you can't have subsubsubentries.

To specify a page range, put an `\index{...|()}` command at the beginning of the range and an `\index{...|)}` command at the end of the range. The two “...”s must be identical.

|                |                                                                     |
|----------------|---------------------------------------------------------------------|
| gnat, vi–x, 22 | Page vi: <code>\index{gnat ()}</code>                               |
| gnus           | Page x: <code>\index{gnat )}</code>                                 |
| bad, 22        | Page 22: <code>\index{gnat}</code>                                  |
| good, 28–32    | <code>\index{gnus!bad ()}</code><br><code>\index{gnus!bad )}</code> |
|                | Page 28: <code>\index{gnus!good ()}</code>                          |
|                | Page 30: <code>\index{gnus!good}</code>                             |
|                | Page 32: <code>\index{gnus!good )}</code>                           |

As the example shows, *MakeIndex* does the right thing when both ends of the range fall on the same page, and when there is an identical entry within the range.

You can add a cross-reference to another entry as follows:

|                         |                                                  |
|-------------------------|--------------------------------------------------|
| at, 2                   | Page 2: <code>\index{at}</code>                  |
| bat, <i>see</i> bat, at | Page 2: <code>\index{at!bat see{bat, at}}</code> |

Since the “*see*” entry does not print a page number, it doesn't matter where the `\index{...|see{...}}` goes, but it must come after the `\begin{document}` command. You might want to put all such cross-referencing commands in one place.

If you specify an entry of the form  $str_1@str_2$ , the string  $str_1$  determines the alphabetical position of the entry, while the string  $str_2$  produces the text of the entry.

|                |                             |
|----------------|-----------------------------|
| twenty, 44     | Page 44: \index{twenty}     |
| xx, 55         | Page 46: \index{twenty-one} |
| twenty-one, 46 | Page 55: \index{twenty@xx}  |

This feature is useful because the argument of the `\index` command provides the actual input string that L<sup>A</sup>T<sub>E</sub>X uses to generate the index entry.

|                  |                                     |
|------------------|-------------------------------------|
| alpha, 13        | Page 12: \index{alphas}             |
| $\alpha$ , 14    | Page 13: \index{alpha}              |
| alphas, 12       | Page 14: \index{alpha@\$\alpha\$}   |
| <b>alps</b> , 33 | Page 33: \index{alps@\textbf{alps}} |

The command `\index{$\alpha$}` will also produce an  $\alpha$  entry in the index, but that entry will be alphabetized as  $\alpha$ .

In some indexes, certain page numbers are specially formatted—for example, an italic page number may indicate the primary reference, and an *n* after a page number may denote that the item appears in a footnote. *MakeIndex* makes it easy to format an individual page number any way you want. For any string of characters *str*, the command `\index{...|str}` produces a page number of the form `\str{n}`. Similarly, the command `\index{...|(str)}` produces a page number of the form `\str{n-m}`, or of the form `\str{n}` if the specified range includes only a single page.

|               |                                      |
|---------------|--------------------------------------|
| gnat, 3, 4n   | Preamble: \newcommand{\nn}{[1]{#1n}} |
| gnu, 5, 44-46 | Page 3: \index{gnat emph}            |
|               | Page 4: \index{gnat nn}              |
|               | Page 5: \index{gnu}                  |
|               | Page 44: \index{gnu (emph)}          |
|               | Page 46: \index{gnu )}               |

The “see” option is a special case of this facility, where the `\see` command is predefined by the `makeidx` package.

### A.2.3 The Fine Print

The argument of an `\index` command must always have matching braces, where the brace in a `\{` or `\}` command counts. Special characters like `\` may appear in the argument only if the `\index` command is not itself contained in the argument of another command. This is most likely to be a problem when indexing items in a footnote. Even in this case, robust commands can be placed in the “`@`” part of an entry, as in `\index{alp@\textit{alp}}`, and fragile commands can be used if protected with the `\protect` command.

*MakeIndex* works as expected when all page numbers are either arabic or lowercase roman numerals; pages numbered with roman numerals are assumed to precede those numbered with arabic numerals. *MakeIndex* can also handle other types of page numbers; consult its documentation for details.

To put a !, @, or | character in an index entry, *quote* it by preceding the character with a ". More precisely, a character is said to be quoted if it follows an unquoted " that is not part of a \" command. A quoted !, @, or | character is treated like an ordinary character rather than having its usual meaning. The " preceding a quoted character is deleted before the entries are alphabetized.

|                |                                     |
|----------------|-------------------------------------|
| exclaim (!), 2 | Page 2: \index{exclaim ("!)}        |
| loudly, 3      | Page 3: \index{exclaim ("!)!loudly} |
| für, 4         | Page 4: \index{fur@f"\{u\}r}        |
| quote ("), 5   | Page 5: \index{quote (\verb+" "+)}  |

*MakeIndex* regards spaces as ordinary characters when alphabetizing the entries and deciding whether two entries are the same. Thus, the commands \index{gnu}, \index{\\_gnu}, and \index{gnu\\_} produce three separate entries, the first appearing near the beginning of the index, since \\_ (space) comes before any letter in *MakeIndex*'s "alphabetical" order. All three entries look the same when printed, since L<sup>A</sup>T<sub>E</sub>X ignores extra spaces in the input. However, since L<sup>A</sup>T<sub>E</sub>X regards multiple spaces as a single space, \index{a\\_space} and \index{a\\_u\\_space} produce identical output on the *idx* file, so they produce a single index entry. Since % is treated as an ordinary character in the argument of an \index command, there is no way to split the argument across lines without inserting a space into the entry.

### A.3 Error Messages

*MakeIndex* prints out the number of lines read and written and how many errors were found. Messages to identify the errors are written on a file with extension *ilg*. There are two phases in which *MakeIndex* can produce error messages: when it is reading the *idx* file, and when it is writing the *ind* file. Each error message prints the nature of the error followed by a line number, identifying where in the file the error occurs. In the reading phase, the line number refers to the *idx* file; in the writing phase, it refers to the *ind* file. The error messages should enable you to figure out what you did wrong.

## APPENDIX B

# The Bibliography Database



As explained in Section 4.3.1, the `\bibliography` command specifies one or more `bib` files—bibliographic database files whose names have the extension `bib`. `BIBTEX` uses the `bib` file(s) to generate a `bbl` file that is read by `\bibliography` to make the bibliography. This appendix explains how to create `bib` files.

Once you learn to use `BIBTEX`, you will find it easier to let `BIBTEX` make your reference list than to do it yourself. Moreover, you will quickly compile a bibliographic database that eliminates almost all the work of making a bibliography. Other people may have `bib` files that you can copy, or there may be a common database that you can use. Ask your friends or check the *Local Guide* to find out what is available. However, remember that you are responsible for the accuracy of the references in your document. Even published references are notoriously unreliable; don't rely on any bibliography information that has not been carefully checked by someone you trust.

## B.1 The Format of the `bib` File

### B.1.1 The Entry Format

A `bib` file contains a series of entries like the following:

```
@BOOK{kn:gnus,
 AUTHOR = "Donald E. Knudson",
 TITLE = "1966 World Gnus Almanac",
 PUBLISHER = {Permafrost Press},
 ADDRESS = {Novosibirsk} }
```

The `@BOOK` states that this is an entry of type *book*. Various entry types are described below. The `kn:gnus` is the *key*, as it appears in the argument of a `\cite` command referring to the entry.

This entry has four *fields*, named `AUTHOR`, `TITLE`, `PUBLISHER`, and `ADDRESS`. The meanings of these and other fields are described below. A field consists of the name, followed by an `=` character with optional space around it, followed by its text. The text of a field is a string of characters, with no unmatched braces, surrounded by either a pair of braces or a pair of `"` characters. (Unlike in `LATEX` input, `\{` and `\}` are considered to be braces with respect to brace matching.) Entry fields are separated from one another, and from the key, by commas. A comma may have optional space around it.

The outermost braces that surround the entire entry may be replaced by parentheses. As in `LATEX` input files, an end-of-line character counts as a space and one space is equivalent to one hundred. Unlike `LATEX`, `BIBTEX` ignores the case of letters in the entry type, key, and field names, so the entry above could have been typed as follows:

```
@Book{GNUS,
 author={Donald E. Knudson},
 title = {"1966 World
 Gnus Almanac", ...})
```

However, the case of letters does matter to L<sup>A</sup>T<sub>E</sub>X, so the key should appear exactly the same in all `\cite` commands in the L<sup>A</sup>T<sub>E</sub>X input file.

The quotes or braces can be omitted around text consisting entirely of numerals. The following two fields are equivalent:

```
Volume = "27" Volume = 27
```

### B.1.2 The Text of a Field

The text of the field is enclosed in braces or double quote characters (""). A part of the text is said to be *enclosed in braces* if it lies inside a matching pair of braces other than the ones enclosing the entire field.

#### Names

The text of an `author` or `editor` field represents a list of names. The bibliography style determines the format in which a name is printed—whether the first name or last name appears first, if the full first name or just the first initial is used, etc. The `bib` file entry simply tells BIBT<sub>E</sub>X what the name is. You should type an author's complete name, exactly as it appears in the cited work, and let the bibliography style decide what to abbreviate.

Most names can be entered in the obvious way, either with or without a comma, as in the following examples.

```
"John Paul Jones" "Jones, John Paul"
" Ludwig van Beethoven" "van Beethoven, Ludwig"
```

Only the second form, with a comma, should be used for people who have last names with multiple parts that are capitalized. For example, Per Brinch Hansen's last name is Brinch Hansen, so his name should be typed with a comma:

```
"Brinch Hansen, Per"
```

If you type "Per Brinch Hansen", BIBT<sub>E</sub>X will think that "Brinch" is his middle name. "van Beethoven" or "de la Madrid" pose no problem because "van" and "de la" are not capitalized.

"Juniors" present a special problem. People with "Jr." in their name generally precede it with a comma. Such a name should be entered as follows:

```
"Ford, Jr., Henry"
```

**BIBTEX** is sometimes confused by characters that are produced by **LATEX** commands—for example, accented characters and characters produced by the commands of Section 3.2.2. It will do the right thing if you put curly braces immediately around a command that produces a character:

```
"Kurt G{\\"o}del" "V. S{\o}rensen" "J. Mart{\'{i}}"
```

If there are multiple authors or editors, their names are separated by the word **and**. A paper written by Alpher, Bethe, and Gamow has the following entry:

```
AUTHOR = "Ralph Alpher and Bethe, Hans and George Gamow"
```

An **and** separates authors' names only if it is not enclosed in braces. If an **author** or **editor** field has more names than you want to type, just end the list of names with **and others**; the standard styles convert this to the conventional "et al."

### **Titles**

The bibliography style determines whether or not a title is capitalized; the titles of books usually are, the titles of articles usually are not. You type a title the way it should appear if it is capitalized.

```
TITLE = "The Agony and the Ecstasy"
```

You should capitalize the first word of the title, the first word after a colon, and all other words except articles and unstressed conjunctions and prepositions. **BIBTEX** will change uppercase letters to lowercase if appropriate. Uppercase letters that should not be changed are enclosed in braces. The following two titles are equivalent; the *A* of *Africa* will not be made lowercase.

```
"The Gnats and Gnus of {Africa}"
"The Gnats and Gnus of {A}frica"
```

### **B.1.3 Abbreviations**

Instead of an ordinary text string, the text of a field can be replaced by an *abbreviation* for it. An abbreviation is a string of characters that starts with a letter and does not contain a space or any of the following ten characters:

```
" # % ' () , = { }
```

The abbreviation is typed in place of the text field, with no braces or quotation marks. If *jgg1* is an abbreviation for

```
Journal of Gnats and Gnus, Series~1
```

then the following are equivalent:

```
Journal = jgg1
Journal = "Journal of Gnats and Gnus, Series~1"
```

Some abbreviations are predefined by the bibliography style. These always include the usual three-letter abbreviations for the month: `jan`, `feb`, `mar`, etc. Bibliography styles may contain abbreviations for the names of commonly referenced journals. Consult your *Local Guide* for a list of the predefined abbreviations for the bibliography styles available on your computer.

You can define your own abbreviations by putting a `@string` command in the `bib` file. The command

```
@string{jgg1 = "Journal of Gnats and Gnus, Series~1"}
```

defines `jgg1` to be the abbreviation assumed in the previous example. Parentheses can be used in place of the outermost braces in the `@string` command, and braces can be used instead of the quotation marks. The text must have matching braces.

The case of letters is ignored in an abbreviation as well as in the command name `@string`, so the command above is equivalent to

```
@STRING{JgG1 = "Journal of Gnats and Gnus, Series~1"}
```

A `@string` command can appear anywhere before or between entries in a `bib` file. However, it must come before any use of the abbreviation, so a sensible place for `@string` commands is at the beginning of the file. You can also put your abbreviations in a separate `bib` file, say `abbrev.bib`, and use the command

```
\bibliography{abbrev, ...}
```

in your document. A `@string` command in a `bib` file takes precedence over a definition made by the bibliography style, so it can be used to change the definition of an abbreviation such as `Feb`.

#### B.1.4 Cross-References

Several cited sources may be part of a larger work—for example, different papers in the same conference proceedings. You can make a single entry for the conference proceedings, and refer to that entry in the entries for the individual papers. Fields that appear in the proceedings' entry need not be duplicated in the papers' entries. However, every required field for a paper must be either in its entry or in the referenced entry.

```

@INPROCEEDINGS(beestly-gnats,
 AUTHOR = "Will D. Beest",
 TITLE = "Gnats in the Gnus",
 PAGES = "47--59",
 CROSSREF = "ope:6cpb")
...
@PROCEEDINGS(ope:6cpb,
 TITLE = "Sixth Conference on Parasites in Bovidae",
 BOOKTITLE = "Sixth Conference on Parasites in Bovidae",
 EDITOR = "Ann T. L. Ope",
 YEAR = 1975)

```

The apparently redundant `BOOKTITLE` field in the proceedings entry is needed to provide the field of that name for the entry of each paper that cross-references it. As explained below, the `TITLE` field is required to produce a reference-list entry for the proceedings; `BIBTeX` ignores the `BOOKTITLE` field when producing such an entry. The reference list made by `BIBTeX` may have an entry for the proceedings that is cited by the entries for the individual papers, even if the proceedings are not explicitly cited in the original document.

A cross-referenced entry like `ope:6cpb` in the example must come after any entries that refer to it.

## B.2 The Entries

### B.2.1 Entry Types

When entering a reference in the database, the first thing to decide is what type of entry it is. No fixed classification scheme can be complete, but `BIBTeX` provides enough entry types to handle almost any reference reasonably well.

References to different types of publications contain different information; a reference to a journal article might include the volume and number of the journal, which is usually not meaningful for a book. Therefore, database entries of different types have different fields. For each entry type, the fields are divided into three classes:

**required** Omitting the field will produce an error message and will occasionally result in a badly formatted bibliography entry. If the required information is not meaningful, you are using the wrong entry type. If the required information is meaningful but not needed—for example, because it is included in some other field—simply ignore the warning that `BIBTeX` generates.

**optional** The field's information will be used if present, but can be omitted without causing any formatting problems. A reference should contain any information that might help the reader, so you should include the optional

field if it is applicable. (A nonstandard bibliography style might ignore an optional field when creating the reference-list entry.)

**ignored** The field is ignored. **BIBTeX** ignores a field that is not required or optional, so you can include any fields you want in a **bib** file entry. It's a good idea to put all relevant information about a reference in its **bib** file entry—even information that may never appear in the bibliography. For example, if you want to keep an abstract of a paper in a computer file, put it in an **abstract** field in the paper's **bib** file entry. The **bib** file is likely to be as good a place as any for the abstract, and it is possible to design a bibliography style for printing selected abstracts.

Misspelling its name will cause a field to be ignored, so check the database entry if relevant information that you think is there does not appear in the reference-list entry.

The following are all the entry types, along with their required and optional fields, that are used by the standard bibliography styles. The meanings of the individual fields are explained in the next section. A particular bibliography style may ignore some optional fields in creating the reference. Remember that, when used in the **bib** file, the entry-type name is preceded by an @ character.

**article** An article from a journal or magazine. Required fields: **author**, **title**, **journal**, **year**. Optional fields: **volume**, **number**, **pages**, **month**, **note**.

**book** A book with an explicit publisher. Required fields: **author** or **editor**, **title**, **publisher**, **year**. Optional fields: **volume** or **number**, **series**, **address**, **edition**, **month**, **note**.

**booklet** A work that is printed and bound, but without a named publisher or sponsoring institution. Required field: **title**. Optional fields: **author**, **howpublished**, **address**, **month**, **year**, **note**.

**conference** The same as **inproceedings**, included for compatibility with older versions.

**inbook** A part of a book, usually untitled; it may be a chapter (or other sectional unit) and/or a range of pages. Required fields: **author** or **editor**, **title**, **chapter** and/or **pages**, **publisher**, **year**. Optional fields: **volume** or **number**, **series**, **type**, **address**, **edition**, **month**, **note**.

**incollection** A part of a book with its own title. Required fields: **author**, **title**, **booktitle**, **publisher**, **year**. Optional fields: **editor**, **volume** or **number**, **series**, **type**, **chapter**, **pages**, **address**, **edition**, **month**, **note**.

**inproceedings** An article in a conference proceedings. Required fields: **author**, **title**, **booktitle**, **year**. Optional fields: **editor**, **volume** or **number**, **series**, **pages**, **address**, **month**, **organization**, **publisher**, **note**.

**manual** Technical documentation. Required field: `title`. Optional fields: `author`, `organization`, `address`, `edition`, `month`, `year`, `note`.

**mastersthesis** A master's thesis. Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

**misc** Use this type when nothing else fits. Required fields: `none`. Optional fields: `author`, `title`, `howpublished`, `month`, `year`, `note`.

**phdthesis** A Ph.D. thesis. Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

**proceedings** The proceedings of a conference. Required fields: `title`, `year`. Optional fields: `editor`, `volume` or `number`, `series`, `address`, `month`, `organization`, `publisher`, `note`.

**techreport** A report published by a school or other institution, usually numbered within a series. Required fields: `author`, `title`, `institution`, `year`. Optional fields: `type`, `number`, `address`, `month`, `note`.

**unpublished** A document with an author and title, but not formally published. Required fields: `author`, `title`, `note`. Optional fields: `month`, `year`.

In addition to the fields listed above, each entry type also has an optional `key` field, used in some styles for alphabetizing and forming a `\bibitem` label. You should include a `key` field for any entry with no `author` or `author substitute`. (Depending on the entry type, an `editor` or an `organization` can substitute for an `author`.) Do not confuse the `key` field with the `key` that appears in the `\cite` command and at the beginning of the whole entry, after the entry type.

## B.2.2 Fields

Below is a description of all the fields recognized by the standard bibliography styles. An entry can also contain other fields that are ignored by those styles.

**address** Usually the address of the `publisher` or `institution`. For major publishing houses, omit it entirely or just give the city. For small publishers, you can help the reader by giving the complete address.

**annote** An annotation. It is not used by the standard bibliography styles, but may be used by other styles that produce an annotated bibliography.

**author** The name(s) of the author(s), in the format described above.

**booktitle** The title of a book, a titled part of which is being cited. It is used only for the `incollection` and `inproceedings` entry types; use the `title` field for `book` entries. How to type titles is explained above.

**chapter** A chapter (or other sectional unit) number.

**crossref** The database key of the entry being cross-referenced.

**edition** The edition of a book—for example, “Second”. (The style will convert this to “second” if appropriate.)

**editor** The name(s) of editor(s), typed as indicated above. If there is also an **author** field, then the **editor** field gives the editor of the book or collection in which the reference appears.

**howpublished** How something strange was published.

**institution** The sponsoring institution of a technical report.

**journal** A journal name. Abbreviations may exist; see the *Local Guide*.

**key** Used for alphabetizing and creating a label when the **author** and **editor** fields are missing. This field should not be confused with the key that appears in the `\cite` command and at the beginning of the entry.

**month** The month in which the work was published or, for an unpublished work, in which it was written. Use the standard three-letter abbreviations described above.

**note** Any additional information that can help the reader. The first word should be capitalized.

**number** The number of a journal, magazine, technical report, or work in a series. An issue of a journal or magazine is usually identified by its volume and number; the organization that issues a technical report usually gives it a number; books in a named series are sometimes numbered.

**organization** The organization that sponsors a conference or that publishes a manual.

**pages** One or more page numbers or ranges of numbers, such as 42--111 or 7,41,73--97.

**publisher** The publisher’s name.

**school** The name of the school where a thesis was written.

**series** The name of a series or set of books. When citing an entire book, the **title** field gives its title and the optional **series** field gives the name of a series or multivolume set in which the book was published.

**title** The work’s title, typed as explained above.

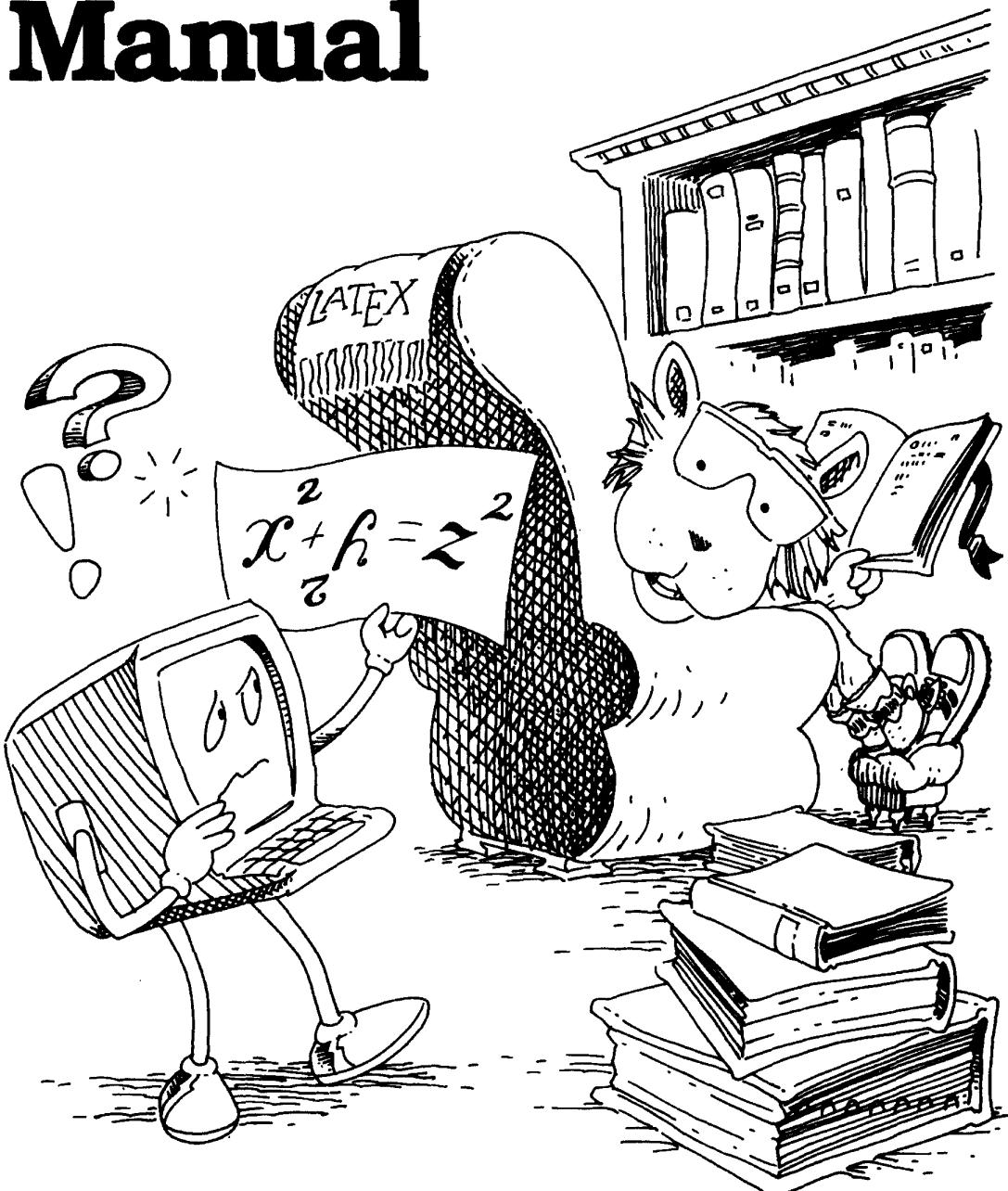
**type** The type of a technical report—for example, “Research Note”. It is also used to specify a type of sectional unit in an `inbook` or `incollection` entry and a different type of thesis in a `mastersthesis` or `phdthesis` entry.

**volume** The volume of a journal or multivolume book.

**year** The year of publication or, for an unpublished work, the year it was written. It usually consists only of numerals, such as 1984, but it could also be something like `circa 1066`.

## APPENDIX C

# Reference Manual



This appendix describes all *L<sup>A</sup>T<sub>E</sub>X* commands and environments, including some features, anomalies, and special cases not mentioned earlier. You should look here when a command or environment does something surprising, or when you encounter a formatting problem not discussed in earlier chapters.

Command and environment descriptions are concise; material explained in an earlier chapter is sketched very briefly. The syntax of commands and environments is indicated by a *command form* such as:

`\parbox[pos]{wdth}{text}`

Everything in typewriter style, such as the `\parbox`[, represents material that appears in the input file exactly as shown. The italicized parts *pos*, *wdth*, and *text* represent items that vary; the command's description explains their function. Arguments enclosed in square brackets [ ] are optional; they (and the brackets) may be omitted, so `\parbox` can also have the form

`\parbox{wdth}{text}`

The case in which an optional argument is missing is called the *default*. If a command form has two optional arguments that come one right after the other, when only one is present it is assumed to be the first one.

A number of *style parameters* are listed in this appendix. Except where stated otherwise, these parameters are length commands. A length is rigid unless it is explicitly said to be a rubber length (Section 6.4.1).

## C.1 Commands and Environments

### C.1.1 Command Names and Arguments

The six commands # \$ & ~ \_ ^ are the only ones with single-character names. The character %, while not a command, causes *L<sup>A</sup>T<sub>E</sub>X* to ignore all characters following it on the input line—including the space character that ends the line—and all space characters at the beginning of the next line. A % can be used to begin a comment and to start a new line without producing space in the output. However, a command name cannot be split across lines.

About two dozen commands have two-character names composed of \ followed by a single nonletter. All other command names consist of \ followed by one or more letters. Command names containing an @ character can be used only in the *sty* files that implement packages (Section 6.1.4). Upper- and lowercase letters are considered to be different, so `\gamma` and `\Gamma` are different commands. Spaces are ignored after a command name of this form, except that a blank line following the command still denotes the end of a paragraph.

Commands may have mandatory and/or optional arguments. A mandatory argument is enclosed by curly braces { and } and an optional argument is enclosed by square brackets [ and ]. Space between arguments is ignored.

The following commands take an optional last argument:

```
\\" \linebreak \nolinebreak \newcounter \twocolumn
\item \pagebreak \nopagebreak \newtheorem \suppressfloats
```

If that argument is missing and the next nonspace character in the text is a [, then L<sup>A</sup>T<sub>E</sub>X will mistake this [ for the beginning of an optional argument. Enclosing the [ in braces prevents this mistake.

Enclosing text in braces can seldom cause trouble.

- [This is an aside.] This is the rest of the item.

```
... \begin{itemize}
```

```
\item {[This is an aside.]} This is ...
```

A ] within the optional argument of an \item command must be enclosed in braces to prevent its being mistaken for the ] that marks the end of the argument.

[gnu] A large animal, found mainly in dictionaries.

```
\begin{description}
```

```
\item {[{[gnu]}] A large animal...}
```

[gnat] A small animal, found mainly in tents.

```
\item {[{[gnat]}] A small animal...}
```

```
\end{description}
```

Some commands, including \\, have a \*-form that is obtained by typing a \* right after the command name. If a \* is the first nonspace character following a command like \\, then it should be enclosed in braces; otherwise, L<sup>A</sup>T<sub>E</sub>X will mistake the \\ and \* for a \\\* command.

## C.1.2 Environments

An environment is begun with a \begin command having the environment's name as the first argument. Any arguments of the environment are typed as additional arguments to the \begin. The environment is ended with an \end command having the environment's name as its only argument. If an environment has a \*-form, the \* is part of the environment's name, appearing in the argument of the \begin and \end commands.

## C.1.3 Fragile Commands

Commands are classified as either *robust* or *fragile*. Type-style-changing commands such as \textbf and \em are robust, as are most of the math-mode commands of Section 3.3. Any command with an optional argument is fragile.

Certain command arguments are called *moving* arguments. A fragile command that appears in a moving argument must be preceded by a \protect command. A \protect applies only to the command it precedes; fragile commands appearing in its argument(s) require their own \protect commands. The following are all the commands and environments with moving arguments:

- Commands with an argument that may be put into a table of contents, list of figures, or list of tables: `\addcontentsline`, `\addtocontents`, `\caption`, and the sectioning commands. If an optional argument is used with a sectioning or `\caption` command, then it is this argument that is the moving one.
- Commands to print on the terminal: `\typeout` and `\typein`. The optional argument of `\typein` is not a moving argument.
- Commands to generate page headings: `\markboth` (both arguments) and `\markright`. (The sectioning commands, already listed, fall under this category too.)
- The `letter` environment (defined in the `letter` document class).
- The `\thanks` command.
- The optional argument of `\bibitem`.
- An `\@` in an `array` or `tabular` environment. (Although `\@` is not a command, fragile commands in an `\@-expression` must be `\protect`'ed as if they were in a moving argument.)

All length commands are robust and must not be preceded by `\protect`. A `\protect` command should not be used in an argument of a `\setcounter` or `\addtocounter` command.

#### C.1.4 Declarations

A declaration is a command that changes the value or meaning of some command or parameter. The *scope* of a declaration begins with the declaration itself and ends with the first `}` or `\end` whose matching `{` or `\begin` occurs before the declaration. The commands `\bgroup`, `\egroup`, and `\$` that end a math-mode environment and the `}` or `]` that end the argument of a L<sup>A</sup>T<sub>E</sub>X command also delimit the scope of a declaration; but the `}` or `]` ending the argument of a command defined with `\newcommand`, or `\renewcommand`, or `\providecommand` does *not* delimit its scope. A declaration is in effect throughout its scope, except within the scope of a subsequent countermanding declaration.

The following declarations are *global*; their scope is not delimited by braces or environments.

|                            |                             |                          |                           |
|----------------------------|-----------------------------|--------------------------|---------------------------|
| <code>\newcounter</code>   | <code>\pagenumbering</code> | <code>\newlength</code>  | <code>\hyphenation</code> |
| <code>\setcounter</code>   | <code>\thispagestyle</code> | <code>\newsavebox</code> |                           |
| <code>\addtocounter</code> | <code>\pagecolor</code>     | <code>\newtheorem</code> |                           |

### C.1.5 Invisible Commands and Environments

A number of commands and environments are “invisible”, meaning that they do not produce any text at the point where they appear.  $\text{\TeX}$  regards an invisible command or environment in the middle of a paragraph as an invisible “word”. Putting spaces or an end-of-line character both before and after an invisible word can generate two separate interword spaces, one on either side of this word, producing extra space in the output. Moreover, if the invisible word occurs at the end of a paragraph, not attached to a real word, it could appear on a line by itself, producing a blank line in the output. Invisible words caused by a command with no argument are seldom a problem, since spaces are ignored when they follow a command name that ends in a letter. Also, the following invisible commands and environments usually eliminate this extra space:<sup>1</sup>

|                           |                           |                         |                     |
|---------------------------|---------------------------|-------------------------|---------------------|
| <code>\pagebreak</code>   | <code>\nolinebreak</code> | <code>\vspace</code>    | <code>\color</code> |
| <code>\nopagebreak</code> | <code>\label</code>       | <code>\glossary</code>  | <code>figure</code> |
| <code>\linebreak</code>   | <code>\index</code>       | <code>\marginpar</code> | <code>table</code>  |

Any other invisible command with an argument that appears inside a paragraph should be attached to an adjacent word, as should the commands and environments listed above in certain unusual situations where they can produce extra space in the output.

### C.1.6 The `\backslash` Command

`\backslash` [*len*]  
`\backslash*` [*len*]

These commands start a new line and add an extra vertical space of length *len* above it. The default is to add no extra space. The `*`-form inhibits a page break before the new line. They may be used in paragraph mode and within the following commands and environments:

|                      |                       |                          |
|----------------------|-----------------------|--------------------------|
| <code>array</code>   | <code>eqnarray</code> | <code>\shortstack</code> |
| <code>tabular</code> | <code>tabbing</code>  | <code>\author</code>     |

$\text{\LaTeX}$  is in paragraph mode, so a `\backslash` can be used, in the following environments (among others):

|                    |                     |                        |                         |
|--------------------|---------------------|------------------------|-------------------------|
| <code>verse</code> | <code>center</code> | <code>flushleft</code> | <code>flushright</code> |
|--------------------|---------------------|------------------------|-------------------------|

and when processing the argument of a `\title`, `\date`, or sectioning command. Do not use two `\backslash` commands in a row in paragraph mode; instead, use an optional argument to add extra vertical space.

In the `array` and `tabular` environments, the spacing between rows is obtained by putting a strut (Section 6.4.3) on each line; a positive value of *len*

---

<sup>1</sup>More precisely, spaces that follow these commands and environments are ignored if there is space in the output before the invisible word that they generate.

increases the depth of this strut. This can fail to add the expected amount of extra space if an object in the row extends further below the line than the default strut.

The `\\"` and `\\"*` commands are fragile.

## C.2 The Structure of the Document

A document consists of the following parts.

### **prepended files**

A possibly empty sequence of `filecontents` environments (Section 4.7).

### **preamble**

Begins with a `\documentclass` command. It may contain `\usepackage` commands, declarations that apply to the entire document, and command and environment definitions.

`\begin{document}`

**text of the document**

`\end{document}`

## C.3 Sentences and Paragraphs

### C.3.1 Making Sentences

The following commands and characters are for use mainly in paragraph and LR mode. They are robust.

#### **quotes**

`'` Apostrophe.    `'text'`    Single quotes.    `“text”`    Double quotes.

#### **dashes**

`-` Intra-word.    `--` Number-range.    `---` Punctuation.

#### **spacing**

`\,` Produces a small space; use it between a double and a single quote.

`\_` Produces an interword space.

`~` Produces an interword space where no line break can occur.

`\@` Causes an “end-of-sentence” space after punctuation when typed before the punctuation character. Needed only if the character preceding the punctuation character is not a lowercase letter or a number.

`\frenchspacing` Suppresses extra space after punctuation, even when `\o` is used. Fragile.

`\nonfrenchspacing` Reverses the effect of `\frenchspacing`. Fragile.

#### special characters

|                    |                     |                |                 |                |                 |                |                 |
|--------------------|---------------------|----------------|-----------------|----------------|-----------------|----------------|-----------------|
| <code>\$</code>    | <code>\\$</code>    | <code>%</code> | <code>\%</code> | <code>{</code> | <code>\{</code> | <code>-</code> | <code>\_</code> |
| <code>&amp;</code> | <code>\&amp;</code> | <code>#</code> | <code>\#</code> | <code>}</code> | <code>\}</code> |                |                 |

(These commands can be used in math mode.) See Sections 3.2 and 3.3.2 for commands to make other symbols.

#### logos

`\LaTeX` Produces `\LaTeX` logo.

`\TeX` Produces `\TeX` logo.

`\today` Generates the current date, in the following format: May 18, 1994.

`\emph{text}` Emphasizes *text*, usually by printing it in italic type.

`\mbox{text}` Typesets *text* in LR mode inside a box, which prevents it from being broken across lines. It can be used in math mode. (See Section 6.4.3.)

### C.3.2 Making Paragraphs

A paragraph is ended by one or more completely blank lines—lines not containing even a `%`. A blank line should not appear where a new paragraph cannot be started, such as in math mode or in the argument of a sectioning command.

`\noindent` When used at the beginning of the paragraph, it suppresses the paragraph indentation. It has no effect when used in the middle of a paragraph. Robust.

`\indent` Produces a horizontal space whose width equals the width of the paragraph indentation. It is used to add a paragraph indentation where one would otherwise be suppressed. Robust.

`\par` Equivalent to a blank line; often used to make command and environment definitions easier to read. Robust.

#### Style Parameters

`\textwidth` Normal width of text on the page. Should be changed only in the preamble.

`\linewidth` Width of lines in the current environment; equals `\textwidth` except when inside a displayed-paragraph environment such as `quote` or `itemize`. Its value should not be changed with the length-setting commands.

**\parindent** Width of the indentation at the beginning of a paragraph. Its value is set to zero in a `parbox`. Its value may be changed anywhere.

**\baselineskip** The minimum space from the bottom of one line to the bottom of the next line in a paragraph. (The space between individual lines may be greater if they contain tall objects.) Its value is set by a type-size-changing command (Section 6.7.1). The value used for the entire paragraph unit (Section 6.2.1) is the one in effect at the blank line or command that ends the paragraph unit. Its value may be changed anywhere.

**\baselinestretch** A decimal number (such as 2 or 1.5). Its default value is 1 and is changed with `\renewcommand`. The value of `\baselineskip` is set by `\begin{document}` and by each type-size-changing command to its default value times `\baselinestretch`. You can produce a “double-spaced” version of the document for copy editing by setting `\baselinestretch` to 2, but it will be ugly and hard to read. Any other changes to the interline spacing should be part of the complete document design, best done by a competent typographic designer.

**\parskip** The extra vertical space inserted before a paragraph. It is a rubber length that usually has a natural length of zero. Its value may be changed anywhere, but should be a stretchable length when a `\flushbottom` declaration (Section 6.1.2) is in effect.

### C.3.3 Footnotes

**\footnote[*num*]{*text*}**

Produces a footnote with *text* as its text and *num* as its number. The *num* argument is a positive integer, even when footnotes are “numbered” with letters or other symbols; if it is missing, then the `footnote` counter is stepped and its value used as the footnote number. This command may be used only in paragraph mode to produce ordinary footnotes. It should not be used inside a box except within a `minipage` environment, in which case it may be used in LR or math mode as well as paragraph mode and the footnote appears at the bottom of the box ended by the next `\end{minipage}`. This may be the wrong place for it if there are nested `minipage` environments. Fragile.

**\footnotemark[*num*]**

Used in conjunction with `\footnotetext` to footnote text where a `\footnote` command cannot be used. It produces a footnote mark (the footnote number that appears in the running text), but it does not produce a footnote. See Figure C.1 for an example of its use. It steps the `footnote` counter if the optional argument is missing. It may be used in any mode. Fragile.

It was **Gnats<sup>12</sup> and Gnus<sup>13</sup>** as we trekked through Africa in the blazing noontime heat.

⋮

<sup>12</sup>Small insects.

<sup>13</sup>Large mammals.

It was `\fbox{Gnats\footnotemark\ and Gnus\footnotemark}\%`  
`\addtocounter{footnote}{-1}\footnotetext{Small insects.}\addtocounter{footnote}{1}\footnotetext{Large mammals.}` as we ...

Figure C.1: Making footnotes without the `\footnote` command.

**`\footnotetext[num]{text}`**

Used in conjunction with `\footnotemark` to footnote text where the `\footnote` command cannot be used. See Figure C.1 for an example. It produces a footnote, just like the corresponding `\footnote` command, except that no footnote mark is generated and the footnote counter is not stepped. Fragile.

### Style Parameters

**`\footnotesep`** The height of a strut placed at the beginning of every footnote to produce the vertical space between footnotes. It may be changed anywhere; the value used is the one in effect when the `\footnote` or `\footnotetext` command is processed.

**`\footnoterule`** A command that draws the line separating the footnotes from the main text. It is used by **L<sup>A</sup>T<sub>E</sub>X** in paragraph mode, between paragraphs (in **T<sub>E</sub>X**'s inner vertical mode). The output it generates must take zero vertical space, so negative space should be used to compensate for the space occupied by the rule. It can be redefined anywhere with `\renewcommand`; the definition used is the one in effect when **T<sub>E</sub>X** produces the page of output.

### C.3.4 Accents and Special Symbols

Commands for making accents in normal text are listed in Table 3.1 on page 38; commands for making accents in math formulas are listed in Table 3.11 on page 50. See Section C.10.1 for commands used in a `tabbing` environment to produce the accents normally made with `\=`, `\'`, and `\``.

Non-English symbols are made with commands listed in Table 3.2 on page 39. The following commands for making additional special symbols can also be used in any mode:

|                      |                   |                           |
|----------------------|-------------------|---------------------------|
| <code>† \dag</code>  | <code>§ \S</code> | <code>© \copyright</code> |
| <code>‡ \ddag</code> | <code>¶ \P</code> | <code>£ \pounds</code>    |

Section 3.3.2 gives many commands for generating symbols in mathematical formulas.

## C.4 Sectioning and Table of Contents

The use of the following commands for producing section headings and table of contents entries is illustrated in Figure C.2.

### C.4.1 Sectioning Commands

`sec_cmd [toc_entry] {heading}`  
`sec_cmd*{heading}`

Commands to begin a sectional unit. The `*`-form suppresses the section number, does not increment the counter, does not affect the running head, and produces no table of contents entry. The `secnumdepth` counter, described below, determines which sectional units are numbered.

`sec_cmd` One of the following:

`\part`      `\section`      `\subsubsection`    `\subparagraph`  
`\chapter`    `\subsection`    `\subsubsection`    `\subparagraph`

Each sectional unit should be contained in the next higher-level unit, except that `\part` is optional. The `article` document class does not have a `\chapter` command.

`toc_entry` Produces the table of contents entry and may be used for the running head (Section 6.1.2). It is a moving argument. If it is missing, the `heading` argument is used for these purposes.

`heading` Produces the section heading. If the `toc_entry` argument is missing, then it is a moving argument that provides the table of contents entry and may be used for the running head (Section 6.1.2).

### Gnats and Gnus Forever

From insects embedded in amber and fossils...

In the table of contents:

|                              |    |
|------------------------------|----|
| Gnats                        | 37 |
| ↓ 2 ex                       |    |
| 2.2x Gnus                    | 37 |
| 2.3 Gnats and Gnus on Gneiss | 37 |

In the text (on page 37):

**2.3 Insects and Ungulates on ...**

`\subsection*{Gnats and Gnus Forever}`

`From insects embedded in amber and ...`

---

`\addcontentsline{toc}{subsection}{Gnats}`

`\addtocontents{toc}{\protect\vspace{2ex}}`

`\addcontentsline{toc}{subsection}{\protect\numberline{2.2x}{Gnus}}`

`\subsection{Gnats and Gnus on Gneiss}`  
`\{Insects and Ungulates on Metamorphic Rock}`

Figure C.2: Sectioning and table of contents commands.

### C.4.2 The Appendix

#### `\appendix`

A declaration that changes the way sectional units are numbered. In the standard `article` document class, appendix sections are numbered “A”, “B”, etc. In the `report` and `book` classes, appendix chapters are numbered “A”, “B”, etc., and the chapter number is printed in the heading as “Appendix A”, “Appendix B”, etc. The `\appendix` command generates no text and does not affect the numbering of parts.

### C.4.3 Table of Contents

```
\tableofcontents
\listoffigures
\listoftables
```

Generate a table of contents, list of figures, and list of tables, respectively. These commands cause L<sup>A</sup>T<sub>E</sub>X to write the necessary information on a file having the same first name as the root file and the following extension:

|                   |                               |                             |                            |
|-------------------|-------------------------------|-----------------------------|----------------------------|
| <i>command:</i>   | <code>\tableofcontents</code> | <code>\listoffigures</code> | <code>\listoftables</code> |
| <i>extension:</i> | <code>toc</code>              | <code>lof</code>            | <code>lot</code>           |

A table of contents or a list of figures or tables compiled from the information on the current version of this file is printed at the point where the command appears.

Table of contents entries are produced by the sectioning commands, and list of figures or tables entries are produced by a `\caption` command in a `figure` or `table` environment (Section 3.5.1). The two commands described below also produce entries.

```
\addcontentsline{file}{sec_unit}{entry}
```

Adds an entry to the specified list or table.

*file* The extension of the file on which information is to be written: `toc` (table of contents), `lof` (list of figures), or `lot` (list of tables).

*sec\_unit* Controls the formatting of the entry. It should be one of the following, depending upon the value of the *file* argument:

- `toc`: the name of the sectional unit, such as `part` or `subsection`
- `lof`: `figure`
- `lot`: `table`

There is no `\` in this argument.

*entry* The text of the entry. It is a moving argument. To produce a line with a sectional-unit or figure or table number, *entry* should be of the form

`\protect\numberline{sec_num}{heading}`

where *sec\_num* is the number and *heading* is the heading.

`\addtocontents{file}{text}`

Adds text (or formatting commands) directly to the file that generates the table of contents or list of figures or tables.

*file* The extension of the file on which information is to be written: `toc` (table of contents), `lof` (list of figures), or `lot` (list of tables).

*text* The information to be written. It is a moving argument.

#### C.4.4 Style Parameters

Parameters control which sectional units are numbered and which are listed in the table of contents. Each sectional unit has a *level number*. In all document classes, sections have level number 1, subsections have level number 2, etc. In the `article` document class, parts have level number 0; in the `report` and `book` classes, chapters have level number 0 and parts have level number -1.

The following two counters (Section 6.3) are provided; they can be set in the preamble.

`secnumdepth` The level number of the least significant sectional unit with numbered headings. A value of 2 means that subsections are numbered but subsubsections are not.

`tocdepth` The level number of the least significant sectional unit listed in the table of contents.

### C.5 Classes, Packages, and Page Styles

#### C.5.1 Document Class

`\documentclass[options]{class}`

Specifies the document class and options.

*class* The document class. The standard classes are: `article`, `report`, `book`, `letter` (for letters), and `slides` (for slides).

*options* A list of one or more options, separated by commas—with no spaces.

The options recognized by the standard document classes are listed below. Alternatives, at most one of which should appear, are separated by the symbol “|”.

**10pt | 11pt | 12pt** Chooses the normal (default) type size of the document.

The default is 10pt, which selects ten-point type. (These options are not recognized by the **slides** class.)

**letterpaper | legalpaper | executivepaper | a4paper | a5paper | b5paper**

Causes the output to be formatted for the appropriate paper size:

|           |                 |    |               |
|-----------|-----------------|----|---------------|
| letter    | 8.5in × 11in    | A4 | 210mm × 297mm |
| legal     | 8.5in × 14in    | A5 | 148mm × 210mm |
| executive | 7.25in × 10.5in | B5 | 176mm × 250mm |

The default is **letterpaper**.

**landscape** Causes the output to be formatted for landscape (sideways) printing on the selected paper size. This option effectively interchanges the width and height dimensions of the paper size.

**final | draft** If TeX has trouble finding good places to break lines, it can produce lines that extend past the right margin (“overfull hboxes”). The **draft** option causes such lines to be marked by black boxes in the output. The **final** option, which does not mark these lines, is the default.

**oneside | twoside** Formats the output for printing on one side or both sides of a page. The default is **oneside**, except that it is **twoside** for the **book** class. (The **twoside** option cannot be used with the **slides** document class.)

**openright | openany** Specifies that chapters must begin on a right-hand page (**openright**) or may begin on any page (**openany**). These options apply only to the **report** class (whose default is **openany**) and the **book** class (whose default is **openright**).

**onecolumn | twocolumn** Specifies one-column or two-column pages. The default is **onecolumn**. (The **twocolumn** option cannot be used with the **letter** or **slides** class.)

**notitlepage | titlepage** The **titlepage** option causes the **\maketitle** command to make a separate title page and the **abstract** environment to put the abstract on a separate page. The default is **titlepage** for all classes except **article**, for which it is **notitlepage**. (These options are not recognized by the **letter** class.)

**openbib** Causes the bibliography (Section 4.3) to be formatted in open style. (This option is not recognized by the **letter** and **slides** classes.)

**leqno** Puts formula numbers on the left side in **equation** and **eqnarray** environments.

**fleqn** Left-aligns displayed formulas.

Putting an option in the `\documentclass` command effectively adds that option to any package (loaded with a `\usepackage` command) that recognizes it. L<sup>A</sup>T<sub>E</sub>X issues a warning message if a document-class option is recognized neither by the document class nor by any loaded package.

### Style Parameters

`\bibindent` Width of the extra indentation of succeeding lines in a bibliography block with the `openbib` style option.

`\columnsep` The width of the space between columns of text in `twocolumn` style.

`\columnseprule` The width of a vertical line placed between columns of text in `twocolumn` style. Its default value is zero, producing an invisible line.

`\mathindent` The amount that formulas are indented from the left margin in the `fleqn` document-class option.

### C.5.2 Packages

`\usepackage[options]{pkgs}`

`pkgs` A list of packages to be loaded. The standard packages include:

`alltt` Defines the `alltt` environment; see Section C.6.4.

`amstex` Defines many commands for mathematical formulas. It is described in the L<sup>A</sup>T<sub>E</sub>X *Companion*.

`babel` For documents in one or more languages other than English; see the L<sup>A</sup>T<sub>E</sub>X *Companion*.

`color` For producing colors, using special device-driver support. A device driver is specified as an option; the *Local Guide* should list the default driver for your computer. See Section 7.3.

`graphics` For geometrical transformations of text and including graphics prepared by other programs. It requires special device-driver support. A device driver is specified as an option; the *Local Guide* should list the default driver for your computer. See Section 7.2.

`graphpap` Defines the `\graphpaper` command for use in the `picture` environment (Section 7.1.4).

`ifthen` Defines simple programming-language constructs (Section C.8.5).

`latexsym` Defines some special mathematical symbols; see Section 3.3.2.

`makeidx` Defines commands for use with *MakeIndex* (Appendix A).

**pict2e** Defines enhanced versions of the `picture` environment commands that remove restrictions on line slope, circle radius, and line thickness.

**showidx** Causes `\index` command arguments to be printed on the page where they occur; see Section 4.5.1.

**options** A list of options, which are provided to all the specified packages. They must be legal options for all the packages.

### C.5.3 Page Styles

An output page consists of a *head*, a *body*, and a *foot*. Style parameters determine their dimensions; the page style specifies the contents of the head and foot. Left-hand and right-hand pages have different parameters. In two-sided style, even-numbered pages are left-hand and odd-numbered pages are right-hand; in one-sided style, all pages are right-hand. All commands described in this section are *fragile*.

#### `\pagestyle{style}`

A declaration, with normal scoping rules, that specifies the current page style. The style used for a page is the one in effect when TEX “cuts the scroll” (page 135). Standard *style* options are:

**plain** The head is empty, the foot has only a page number. It is the default page style.

**empty** The head and foot are both empty.

**headings** The head contains information determined by the document class (usually a sectional-unit heading) and the page number; the foot is empty.

**myheadings** Same as **headings**, except head information (but not the page number) is specified by `\markboth` and `\markright` commands, described below.

#### `\thispagestyle`

Same as `\pagestyle` except it applies only to the current page (the next one to be “cut from the scroll”). It is a global declaration (Section C.1.4).

`\markright{right_head}`  
`\markboth{left_head}{right_head}`

These commands specify the following heading information for the **headings** and **myheadings** page styles:

**left-hand page** Specified by *left\_head* argument of the last `\markboth` before the end of the page.

**right-hand page** Specified by *right\_head* argument of the first `\markright` or `\markboth` on the page, or if there is none, by the last one before the beginning of the page.

Both *right\_head* and *left\_head* are moving arguments. In the `headings` page style, sectioning commands set the page headings with the `\markboth` and `\markright` commands as follows:

| Document Class |                                    |                                         |                          |
|----------------|------------------------------------|-----------------------------------------|--------------------------|
| Printing Style | Command                            | <code>book</code> , <code>report</code> | <code>article</code>     |
| two-sided      | <code>\markboth<sup>a</sup></code> | <code>\chapter</code>                   | <code>\section</code>    |
|                | <code>\markright</code>            | <code>\section</code>                   | <code>\subsection</code> |
| one-sided      | <code>\markright</code>            | <code>\chapter</code>                   | <code>\section</code>    |

<sup>a</sup>Specifies an empty right head.

These commands are overridden as follows:

`\markboth` Put a `\markboth` command right after the sectioning command.

`\markright` Put a `\markright` command immediately before and after the sectioning command, but omit the first one if the sectional unit starts a new page.

The right head information is always null for the first page of a document. If this is a problem, generate a blank first page with the `titlepage` environment.

### `\pagenumbering{num_style}`

Specifies the style of page numbers and sets the value of the `page` counter to 1. It is a global declaration (Section C.1.4). Possible values of *num\_style* are:

`arabic` Arabic numerals.

`roman` Lowercase roman numerals.

`Roman` Uppercase roman numerals.

`alph` Lowercase letters.

`Alpha` Uppercase letters.

The `\pagenumbering` command redefines `\thepage` to be `\num_style{page}`.

### `\twocolumn[text]`

Starts a new page by executing `\clearpage` (Section 6.2.2) and begins typesetting in two-column format. If the *text* argument is present, it is typeset in a double-column-wide parbox at the top of the new page. Fragile.

```
\onecolumn
```

Starts a new page by executing `\clearpage` (Section 6.2.2) and begins typesetting in single-column format. Fragile.

## Style Parameters

Most of the parameters controlling the page style are shown in Figure C.3, where the outer rectangle represents the physical page. These parameters are all rigid lengths. They are normally changed only in the preamble. Anomalies may occur if they are changed in the middle of the document. Odd-numbered pages use `\oddsidemargin` and even-numbered pages use `\evensidemargin`. Not shown in the figure is the parameter `\topskip`, the minimum distance from the top of the body to the reference point of the first line of text. It acts like `\baselineskip` for the first line of a page.

### C.5.4 The Title Page and Abstract

```
\maketitle
```

Generates a title. With the `notitlepage` document-class option (the default for the `article` class), it puts the title at the top of a new page and issues a `\thispagestyle{plain}` command. With the `titlepage` option (the default for other classes), it puts the title on a separate page, using the `titlepage` environment. Information used to produce the title is obtained from the following declarations; an example of their use is given in Figure C.4. It's best to put these declarations in the preamble.

`\title{text}` Declares *text* to be the title. You may want to use `\\\` to tell L<sup>A</sup>T<sub>E</sub>X where to start a new line in a long title.

`\author{names}` Declares the author(s), where *names* is a list of authors separated by `\and` commands. Use `\\\` to separate lines within a single author's entry—for example, to give the author's institution or address.

`\date{text}` Declares *text* to be the document's date. With no `\date` command, the current date is used.

The arguments of these three commands may include the following command:

`\thanks{text}` Produces a footnote to the title. The *text* is a moving argument. Can be used for an acknowledgment of support, an author's address, etc. The footnote marker is regarded as having zero width, which is appropriate when it comes at the end of a line; if the marker comes in the middle of a line, add extra space with `\_` after the `\thanks` command.

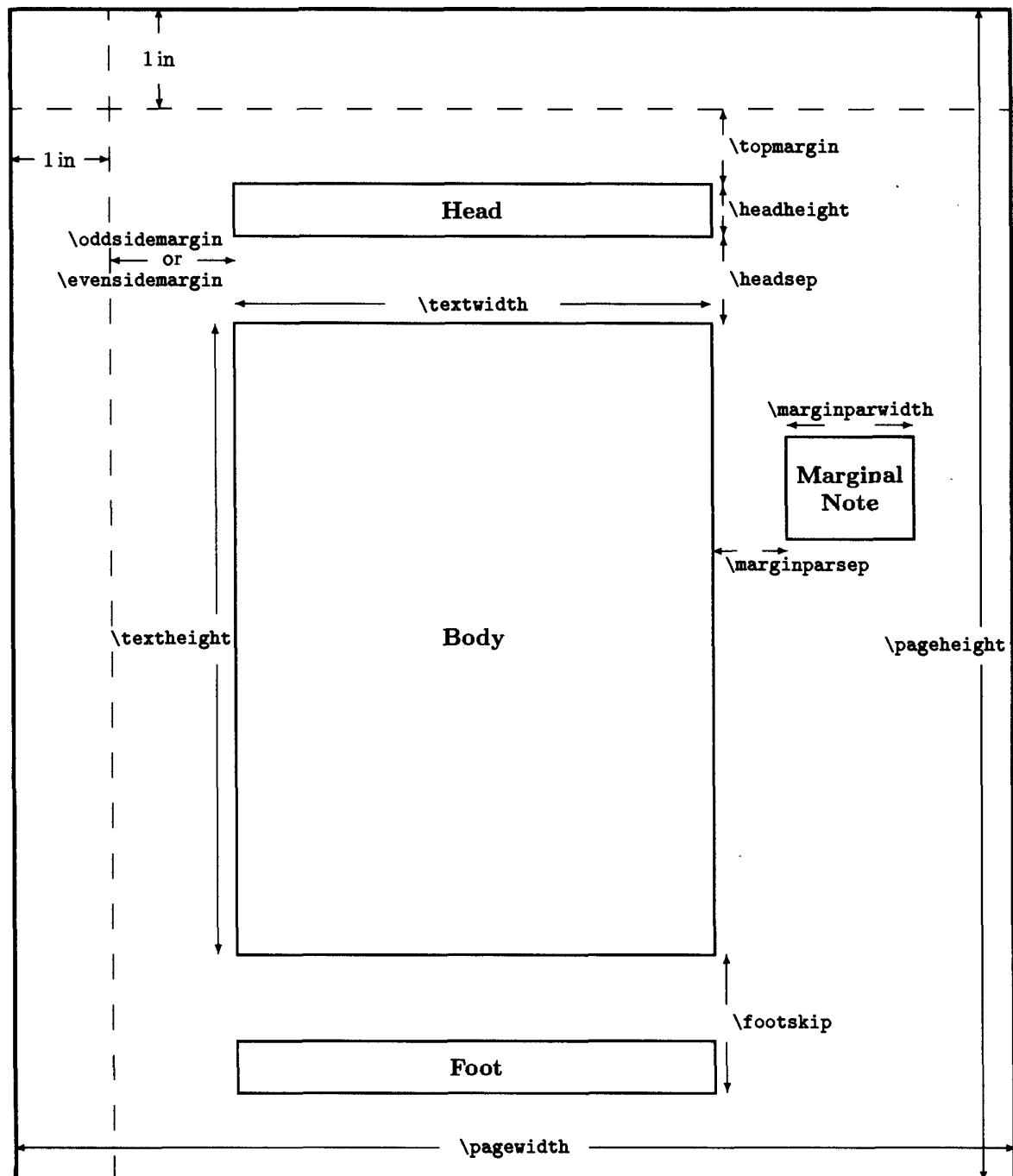


Figure C.3: Page style parameters.

```

Gnu Veldt Cuisine \title{Gnu Veldt Cuisine}

G. Picking* \author{G. Picking\thanks{Supported
Acme Kitchen Products by a grant from the GSF.} \\
 R. Dillo \\
 Cordon Puce School\thanks{On leave during 1985.}}
 \and
 R. Dillo \\ Cordon Puce
 School\thanks{On leave during 1985.}

 :
 \date{24 July 1984 \\
 Revised 5 January 1985}

 ...
 \maketitle

```

\*Supported by a grant from the GSF.  
†On leave during 1985.

Figure C.4: An example title.

```
\begin{abstract} text \end{abstract}
```

Generates an abstract, with *text* as its contents, right where the environment occurs. The abstract is placed on a page by itself when the **titlepage** document-class option (the default for the **report** class; see Section 6.1.1) is in effect. This environment is not defined in the **book** document class.

```
\begin{titlepage} text \end{titlepage}
```

Produces a title page with the **empty** page style and resets the number of the following page to one. You are completely responsible for formatting the contents of this page.

## C.6 Displayed Paragraphs

The output produced by a displayed-paragraph environment starts on a new line, as does the output produced by the text following it. In addition to the environments described in this section, the **tabbing**, **center**, **flushleft**, and **flushright** environments and the environments defined by **\newtheorem** (Section 3.4.3) are also displayed-paragraph environments.

The text following a displayed-paragraph environment begins a new paragraph if there is a blank line after the **\end** command. However, even with no blank line, the following text may have a paragraph indentation if a right brace or **\end** command comes between it and the environment's **\end** command. This anomalous indentation is eliminated with a **\noindent** command (Section C.3.2).

Anomalous extra vertical space may be added after a displayed-paragraph environment that ends with a displayed equation (one made with the **displaymath**,

equation, or `eqnarray` environment). This space can be removed by adding a negative vertical space with a `\vspace` command (Section 6.4.2). You can determine how much space to remove by trial and error.

All displayed-paragraph environments are implemented with the `list` or `trivlist` environment. These environments and the relevant formatting parameters are described in Section C.6.3 below.

### C.6.1 Quotations and Verse

`\begin{quote} text \end{quote}`

For a short quotation or a sequence of short quotations separated by blank lines.

`\begin{quotation} text \end{quotation}`

For a multiparagraph quotation.

`\begin{verse} text \end{verse}`

For poetry. Lines within a stanza are separated by `\backslash` commands and stanzas are separated by one or more blank lines.

### C.6.2 List-Making Environments

`\begin{itemize} item_list \end{itemize}`  
`\begin{enumerate} item_list \end{enumerate}`  
`\begin{description} item_list \end{description}`

The `item_list` consists of a sequence of items, each one begun with an `\item` command (see below). Numbering in an `enumerate` environment is controlled by the counter `enumi`, `enumii`, `enumiii`, or `enumiv`, depending upon its nesting level within other `enumerate` environments. The printed value of this counter is declared to be the current `\ref` value (Section C.11.2).

The default labels of an `itemize` environment are produced by the command `\labelitemi`, `\labelitemii`, `\labelitemiii`, or `\labelitemiv`, depending upon its nesting level within other `itemize` environments. The “tick marks” produced by the `itemize` environment may be changed by redefining these commands with `\renewcommand`.

If an item of a `description` environment begins with a displayed-paragraph environment, the item label may overprint the first line of that environment. If this happens, the item should begin with an `\mbox{}` command to cause the environment to start on a new line.

`\item[label]`

Starts a new item and produces its label. The item label is generated by the `label` argument if present; otherwise the default label is used. In `itemize` and

`enumerate`, the label is typeset flush right a fixed distance from the item's left margin. In `enumerate`, the optional argument suppresses the incrementing of the enumeration counter. The default label is null in the `description` environment. The `\item` command is fragile.

### C.6.3 The list and trivlist Environments

```
\begin{list}{default_label}{decls} item_list \end{list}
```

Produces a list of labeled items.

*item\_list* The text of the items. Each item is begun with an `\item` command (Section C.6.2).

*default\_label* The label generated by an `\item` command with no optional argument.

*decls* A sequence of declarations that can be used to change any of the following parameters that control formatting in the `list` environment. (See also Figure 6.3 on page 113.)

`\topsep` The amount of extra vertical space (in addition to `\parskip`) inserted between the preceding text and the first list item, and between the last item and the following text. It is a rubber length.

`\partopsep` The extra vertical space (in addition to  $\topsep + \parskip$ ) inserted, if the environment is preceded by a blank line, between the preceding text and the first list item and between the last item and the following text. It is a rubber length.

`\itemsep` The amount of extra vertical space (in addition to `\parsep`) inserted between successive list items. It is a rubber length.

`\parsep` The amount of vertical space between paragraphs within an item. It is the value to which `\parskip` is set within the list. It is a rubber length.

`\leftmargin` The horizontal distance between the left margin of the enclosing environment and the left margin of the list. It must be nonnegative. In the standard document classes, it is set to `\leftmargini`, `\leftmarginii`, ... or `\leftmarginvi`, depending on the nesting level of the `list` environment.

`\rightmargin` The horizontal distance between the right margin of the enclosing environment and the right margin of the list. It must be nonnegative. Its default value is zero in the standard document classes.

`\listparindent` The amount of extra indentation added to the first line of every paragraph except the first one of an item. Its default value is zero in the standard document classes. It may have a negative value.

**\itemindent** The indentation of the first line of an item. Its default value is zero in the standard document classes. It may have a negative value.

**\labelsep** The space between the end of the box containing the label and the text of the first line of the item. It may be set to a negative length.

**\labelwidth** The normal width of the box that contains the label. It must be nonnegative. In the standard document styles, its default value is `\leftmargin... - \labelsep`, so the left edge of the label box is flush with the left margin of the enclosing environment. If the natural width of the label is greater than **\labelwidth**, then the label is typeset in a box with its natural width, so the label extends further to the right than “normal”.

**\makelabel{label}** A command that generates the label printed by the **\item** command from the *label* argument. Its default definition positions the label flush right against the right edge of its box. It may be redefined with **\renewcommand**.

In addition to declarations that set these parameters, the following declaration may appear in *decls*:

**\usecounter{ctr}** Enables the counter *ctr* (Section 6.3) to be used for numbering list items. It causes *ctr* to be initialized to zero and incremented by **\refstepcounter** when executing an **\item** command that has no optional argument, causing its value to become the current **\ref** value (Section C.11.2). It is a fragile command.

**\begin{trivlist} *item\_list* \end{trivlist}**

Acts like a *list* environment using the current values of the list-making parameters, except with **\parsep** set to the current value of **\parskip** and the following set to zero: **\leftmargin**, **\labelwidth**, and **\itemindent**. The *trivlist* environment is normally used to define an environment consisting of a single list item, with an **\item** command appearing as part of the environment’s definition.

#### C.6.4 Verbatim

**\begin{verbatim} *literal\_text* \end{verbatim}**  
**\begin{verbatim\*} *literal\_text* \end{verbatim\*}**

Typesets *literal\_text* exactly as typed, including special characters, spaces, and line breaks, using a typewriter type style. The only text following the **\begin** command that is not treated literally is the **\end** command. The *\*-form* differs only in that spaces are printed as `\` symbols.

If there is no nonspace character on the line following the `\begin` command, then *literal\_text* effectively begins on the next line. There can be no space between the `\end` and the `{verbatim}` or `{verbatim*}`.

A `verbatim` or `verbatim*` environment may not appear in the argument of any command.

```
\verbchar literal_text char
\verb*char literal_text char
```

Typesets *literal\_text* exactly as typed, including special characters and spaces, using a typewriter type style. There may be no space between `\verb` or `\verb*` and *char*. The `*-form` differs only in that spaces are printed as `\_` symbols.

*char* Any nonspace character, except it may not be a `*` for `\verb`.

*literal\_text* Any sequence of characters not containing an end-of-line character or *char*.

A `\verb` or `\verb*` command may not appear in the argument of any other command.

```
\begin{alltt} literal_text \end{alltt}
```

Like the `verbatim` environment, except the three characters `\`, `{`, and `}` have their usual meanings. Thus, you can use commands like `\input` in the environment. This command is defined by the `alltt` package.

## C.7 Mathematical Formulas

Unless otherwise noted, all commands described in this section can be used only in math mode. See Section 3.3.8 for an explanation of the display and text math styles.

### C.7.1 Math Mode Environments

```
$ formula $
\(formula \)
\begin{math} formula \end{math}
```

These equivalent forms produce an in-text formula by typesetting *formula* in math mode using text style. They may be used in paragraph or LR mode. The `\(` and `\)` commands are fragile; `$` is robust.

```
\ensuremath{formula}
```

Equivalent to `\(formula\)` when used in paragraph or LR mode; equivalent to *formula* when used in math mode.

---

```
\[formula \]
\begin{displaymath} formula \end{displaymath}
```

These equivalent forms produce a displayed formula by typesetting *formula* in math mode using display style. They may be used only in paragraph mode. The displayed formula is centered unless the `fleqn` document-class option is used (Section 6.1.1). The commands `\[` and `\]` are fragile.

```
\begin{equation} formula \end{equation}
```

The same as `displaymath` except that an equation number is generated using the `equation` counter. The equation number is positioned flush with the right margin, unless the `leqno` document-class option is used (Section 6.1.1).

```
\begin{eqnarray} eqns \end{eqnarray}
\begin{eqnarray*} eqns \end{eqnarray*}
```

Produces a sequence of displayed formulas aligned in three columns. The *eqns* text is like the body of an `array` environment (Section 3.3.3) with argument `rcl`; it consists of a sequence of rows separated by `\backslash` commands, each row consisting of three columns separated by `&` characters. (However, a `\multicolumn` command may not be used.) The first and third columns are typeset in display style, the second in text style. These environments may be used only in paragraph mode.

The `eqnarray` environment produces an equation number for each row, generated from the `equation` counter and positioned as in the `equation` environment. A `\label` command anywhere within the row refers to that row's number. A `\nonumber` command suppresses the equation number for the row in which it appears. The `eqnarray*` environment produces no equation numbers.

The command `\lefteqn{formula}` prints *formula* in display math style (Section 3.3.8), but pretends that it has zero width. It is used within an `eqnarray` or `eqnarray*` environment for splitting long formulas across lines.

An overfull `\hbox` warning occurs if a formula extends beyond the prevailing margins. However, if the formula does lie within the margins, no warning is generated even if it extends far enough to overprint the equation number.

### Style Parameters

`\jot` The amount of extra vertical space added between rows in an `eqnarray` or `eqnarray*` environment.

`\mathindent` The indentation from the left margin of displayed formulas in the `fleqn` document-class option.

`\abovedisplayskip` The amount of extra space left above a long displayed formula—except in the `fleqn` document-class option, where `\topsep` is

used. A long formula is one that lies closer to the left margin than does the end of the preceding line. It is a rubber length.

**\belowdisplayskip** The amount of extra space left below a long displayed formula—except in the **fleqn** document-class option, where **\topsep** is used. It is a rubber length.

**\abovedisplayshortskip** The amount of extra space left above a short displayed formula—except in the **fleqn** document-class option, which uses **\topsep**. A short formula is one that starts to the right of where the preceding line ends. It is a rubber length.

**\belowdisplayshortskip** The amount of extra space left below a short displayed formula—except in the **fleqn** document-class option, which uses **\topsep**. It is a rubber length.

## C.7.2 Common Structures

**\_***{sub}* Typesets *sub* as a subscript. Robust.

**^***{sup}* Typesets *sup* as a superscript. Robust.

**'** Produces a prime symbol (''). Robust.

**\frac***{numer}**{denom}* Generates a fraction with numerator *numer* and denominator *denom*. Robust.

**\sqrt***[n]**{arg}* Generates the notation for the *n*<sup>th</sup> root of *arg*. With no optional argument, it produces the square root (no indicated root). Fragile.

**ellipsis** The following commands produce an ellipsis (three dots) arranged as indicated. They are all robust.

**\ldots** Horizontally at the bottom of the line (...). It may be used in paragraph and LR mode as well as math mode.

**\cdots** Horizontally at the center of the line (· · ·).

**\vdots** Vertically (⋮).

**\ddots** Diagonally (⋱).

## C.7.3 Mathematical Symbols

See Tables 3.3 through 3.8 on pages 41–44. The symbols in Table 3.8 are printed differently in display and text styles; in display style, subscripts and superscripts

may be positioned directly above and below the symbol. All the commands listed in those tables are robust.

Log-like functions, which are set in roman type, are listed in Table 3.9 on page 44. Subscripts appear directly below the symbol in display style for `\det`, `\gcd`, `\inf`, `\lim`, `\liminf`, `\limsup`, `\max`, `\min`, `\Pr`, and `\sup`. All log-like commands are robust. The following commands also create symbols:

`\bmod` Produces a binary *mod* symbol. Robust.

`\pmod{arg}` Produces “(mod *arg*)”. Robust.

## C.7.4 Arrays

See Section C.10.2.

## C.7.5 Delimiters

`\leftdelim formula \rightdelim`

Typesets *formula* and puts large delimiters around it, where *delim* is one of the delimiters in Table 3.10 on page 47 or a ‘.’ character to signify an invisible delimiter. The `\left` and `\right` commands are robust.

## C.7.6 Putting One Thing Above Another

`\overline{formula}`

Typesets *formula* with a horizontal line above it. Robust.

`\underline{formula}`

Typesets *formula* with a horizontal line below it. May be used in paragraph or LR mode as well as math mode. Fragile.

### accents

Table 3.11 on page 50 lists math-mode accent-making commands. They are robust, as are the following additional accenting commands:

`\widehat` Wide version of `\hat`.

`\widetilde` Wide version of `\tilde`.

`\imath` Dotless *i* for use with accents.

`\jmath` Dotless *j* for use with accents.

---

`\stackrel{top}{bot}`

Typesets *top* immediately above *bot*, using the same math style for *top* as if it were a superscript.

### C.7.7 Spacing

The following commands produce horizontal space in math mode. They are all robust. The `\`, command may also be used in paragraph and LR mode.

|                                     |                              |
|-------------------------------------|------------------------------|
| <code>\,</code> thin space          | <code>\:</code> medium space |
| <code>\!</code> negative thin space | <code>\;</code> thick space  |

### C.7.8 Changing Style

#### Type Style

The following commands cause letters, numbers, and uppercase Greek letters in their argument *arg* to be typeset in the indicated style. The commands may be used only in math mode. They are robust.

|                            |          |                             |                     |
|----------------------------|----------|-----------------------------|---------------------|
| <code>\mathrm{text}</code> | Roman    | <code>\mathsf{text}</code>  | Sans Serif          |
| <code>\mathit{text}</code> | Italic   | <code>\mathtt{text}</code>  | Typewriter          |
| <code>\mathbf{text}</code> | Boldface | <code>\mathcal{text}</code> | <i>CALLIGRAPHIC</i> |

The argument of `\mathcal` may contain only uppercase letters.

The following two commands affect all characters in a formula. They may be used only in paragraph or LR mode. They are robust.

`\boldmath` Causes math formulas to be typeset in a bold style. It causes L<sup>A</sup>T<sub>E</sub>X to generate a warning if a formula could use a bold-series font that is not available, even if the font is not actually used.

`\unboldmath` Causes math formulas to be typeset in a normal (nonbold) style.

#### Math Style

The following declarations can appear only in math mode. They choose the type size and certain formatting parameters, including ones that control the placement of subscripts and superscripts. All are robust commands.

`\displaystyle` Default style for displayed formulas.

`\textstyle` Default style for in-text formulas and for the items in an `array` environment.

`\scriptstyle` Default style for first-level subscripts and superscripts.

`\scriptscriptstyle` Default style for higher-level subscripts and superscripts.

## C.8 Definitions, Numbering, and Programming

### C.8.1 Defining Commands

```
\newcommand {cmd}[args][opt]{def}
\renewcommand {cmd}[args][opt]{def}
\providecommand{cmd}[args][opt]{def}
```

These commands define or redefine a command. They are all fragile.

*cmd* A command name beginning with `\` and followed by either a sequence of letters or a single nonletter. For `\newcommand` it must not be already defined and must not begin with `\end`; for `\renewcommand` it must already be defined. The command `\providecommand` acts like `\newcommand` unless *cmd* is already defined, in which case it does nothing and the old definition is retained.

*args* An integer from 1 to 9 denoting the number of arguments of the command being defined. The default is for the command to have no arguments.

*opt* If this argument is present, then the first of the *args* arguments of *cmd* is optional and has a default value of *opt*. If this argument is absent, then all arguments of *cmd* are mandatory.

*def* The text to be substituted for every occurrence of *cmd*; a parameter of the form `#n` in *cmd* is replaced by the text of the *n*<sup>th</sup> argument when this substitution takes place. It may contain command- and environment-defining commands only if all commands and environments they define have no arguments.

The argument-enclosing braces of a command defined or redefined with any of these three commands do not delimit the scope of a declaration in that argument. (However, the scope may be delimited by braces that appear within *def*.) The defined command is fragile if it has an optional argument or if *def* includes a fragile command; otherwise it is robust.

### C.8.2 Defining Environments

```
\newenvironment {nam}[args][opt]{begdef}{enddef}
\renewenvironment{nam}[args][opt]{begdef}{enddef}
```

These commands define or redefine an environment. They are both fragile.

*nam* The name of the environment; it can be any sequence of letters, numbers, and the character `*` that does not begin with `end`. For `\newenvironment`, neither an environment by that name nor the command `\nam` may already be defined. For `\renewenvironment`, the environment must already be defined.

*args* An integer from 1 to 9 denoting the number of arguments of the newly defined environment. The default is no arguments.

*opt* If this argument is present, then the first of the *args* arguments of the environment is optional and has a default value of *opt*. If this argument is absent, then all arguments of the environment are mandatory.

*begdef* The text substituted for every occurrence of `\begin{nam}`; a parameter of the form `#n` in *begdef* is replaced by the text of the *n*<sup>th</sup> argument of `\begin{nam}` when this substitution takes place.

*enddef* The text substituted for every occurrence of `\end{nam}`. It may not contain any argument parameters.

The *begdef* and *enddef* arguments may contain command- and environment-defining commands only if all commands and environments they define have no arguments. The argument-enclosing braces of an environment that was defined with `\newenvironment` or `\renewenvironment` do not delimit the scope of a declaration contained in the argument.

### C.8.3 Theorem-like Environments

```
\newtheorem {env_name}{caption}[within]
\newtheorem {env_name}[numbered_like]{caption}
```

This command defines a theorem-like environment. It is a global declaration (Section C.1.4) and is fragile.

*env\_name* The name of the environment—a string of letters. It must not be the name of an existing environment or counter.

*caption* The text printed at the beginning of the environment, right before the number.

*within* The name of an already defined counter, usually of a sectional unit. If this argument is present, the command `\theenv_name` is defined to be

```
\thewithin.\arabic{env_name}
```

and the *env\_name* counter will be reset by a `\stepcounter{within}` or `\refstepcounter{within}` command (Section C.8.4). If the *within* argument is missing, `\theenv_name` is defined to be `\arabic{env_name}`.

*numbered\_like* The name of an already defined theorem-like environment. If this argument is present, the *env\_name* environment will be numbered in the same sequence (using the same counter) as the *numbered\_like* environment and will declare the current `\ref` value (Section C.11.2) to be the text generated by `\thenumbered_like`.

Unless the *numbered like* argument is present, this command creates a counter named *env\_name*, and the environment declares the current `\ref` value (Section C.11.2) to be the text generated by `\theenv_name`.

The `\newtheorem` command may have at most one optional argument. See Section C.1.1 if a `\newtheorem` without a final optional argument is followed by a `[` character.

### C.8.4 Numbering

The following commands manipulate counters. There are packages that provide additional commands for performing arithmetic calculations; check the *L<sup>A</sup>T<sub>E</sub>X Companion*.

**`\newcounter{newctr}[within]`**

Defines a new counter named *newctr* that is initialized to zero, with `\thenewctr` defined to be `\arabic{newctr}`. It is a global declaration. The `\newcounter` command may not be used in an `\include`'d file (Section 4.4). Fragile.

*newctr* A string of letters that is not the name of an existing counter.

*within* The name of an already defined counter. If this argument is present, the *newctr* counter is reset to zero whenever the *within* counter is stepped by `\stepcounter` or `\refstepcounter` (see below).

**`\setcounter{ctr}{num}`**

Sets the value of counter *ctr* to *num*. It is a global declaration (Section C.1.4). Fragile.

**`\addtocounter{ctr}{num}`**

Increments the value of counter *ctr* by *num*. It is a global declaration (Section C.1.4). Fragile.

**`\value{ctr}`**

Produces the value of counter *ctr*. It is used mainly in the *num* argument of a `\setcounter` or `\addtocounter` command—for example, the command `\setcounter{bean}{\value{page}}` sets counter *bean* equal to the current value of the *page* counter. However, it can be used anywhere that L<sup>A</sup>T<sub>E</sub>X expects a number. The `\value` command is robust, and must never be preceded by a `\protect` command.

#### numbering commands

The following commands print the value of counter *ctr* in the indicated format. They are all robust.

`\arabic{ctr}` Arabic numerals.

`\roman{ctr}` Lowercase roman numerals.

`\Roman{ctr}` Uppercase roman numerals.

`\alph{ctr}` Lowercase letters. The value of *ctr* must be less than 27.

`\Alpha{ctr}` Uppercase letters. The value of *ctr* must be less than 27.

`\fnsymbol{ctr}` Produces one of the nine “footnote symbols” from the following sequence: \* † ‡ § ¶ || \*\* †† ‡‡. It may be used only in math mode. The value of *ctr* must be less than 10.

`\thectr`

A command used to print the value associated with counter *ctr*. Robust.

`\stepcounter {ctr}`  
`\refstepcounter{ctr}`

Increment the value of counter *ctr* by one and reset the value of any counter numbered “within” it. For example, the `subsection` counter is numbered within the `section` counter, which, in the `report` or `book` document style, is numbered within the `chapter` counter. The `\refstepcounter` command also declares the current `\ref` value (Section C.11.2) to be the text generated by `\thectr`.

## C.8.5 The `ifthen` Package

The `ifthen` package provides commands for writing simple programs with tests and loops. The use of these commands is illustrated in Figure C.5. This section may not make much sense unless you have already done some programming.

`\ifthenelse{test}{then\_txt}{else\_txt}`

If *test* is true, then *then\\_txt* is processed; if *test* is false, then *else\\_txt* is processed. The *then\\_txt* and *else\\_txt* arguments can be any `LATEX` input. The *test* argument must be an expression that `LATEX` evaluates to *true* or *false*. It can be any of the following:

*num<sub>1</sub>* *op* *num<sub>2</sub>* A numerical relation, where *op* is one of the following three characters: > = <, and *num<sub>1</sub>* and *num<sub>2</sub>* are numbers. For example, `\value{page}>17` evaluates to *true* if and only if the current value of the `page` counter is greater than 17.

`\equal{str1}{str2}` Evaluates to *true* if and only if `TEX` regards *str<sub>1</sub>* and *str<sub>2</sub>* as equal. `TEX` may think the two arguments are different even if they print the same—for example, `\today` and `May 1, 2001` are not equal, even on

The gcd (greatest common divisor) of  $m$  and  $n$  is printed by the following algorithm, which successively subtracts the smaller value from the larger until the two are equal.

```

a := m; b := n;
while a ≠ b
 do if a > b then a := a - b
 else b := b - a ;
print(a)

```

For example:  $\text{Gcd}(54,30) = \text{gcd}(24,30) = \text{gcd}(24,6) = \text{gcd}(18,6) = \text{gcd}(12,6) = \text{gcd}(6,6) = 6$ .

```

\newcounter{ca} \newcounter{cb}
\newcommand{\printgcd}[2]{%
 \setcounter{ca}{#1}\setcounter{cb}{#2}%
 \Gcd(#1,#2) =
 \whiledo{\not{(\value{ca}=\value{cb})}}{%
 \ifthenelse{(\value{ca}>\value{cb})}{%
 \addtocounter{ca}{-\value{cb}}%
 \addtocounter{cb}{-\value{ca}}%
 \Gcd(\arabic{ca},\arabic{cb}) = }{%
 \arabic{ca}.}%
 ... For example: \printgcd{54}{30}
 }

```

Figure C.5: Writing programs with the `ifthen` package's commands.

May 1, 2001.  $\text{\TeX}$  will think  $str_1$  and  $str_2$  are equal if replacing every command by its definition makes them identical. For example,  $\text{no\_g}\{x\}$  and  $\text{no\_x-gnu-x}$  are equal if  $\text{g}$  is defined by

```
\newcommand{\g}[1]{#1-gnu-#1}
```

If you're not sure exactly what equals what, try some experiments.

`\lengthtest{len1 op len2}` A length relation, where *op* is one of the following three characters:  $>$  =  $<$ , and *len<sub>1</sub>* and *len<sub>2</sub>* are lengths. For example, `\lengthtest{\parindent < 1cm}` evaluates to *true* if and only if the current value of `\parindent` is less than 1 centimeter. In evaluating the relation, a rubber length is replaced by its natural length (Section 6.4.1).

`\isodd{num}` Evaluates to *true* if and only if the number *num* is odd. It is used to produce different text for left- and right-hand pages. However, the obvious `\isodd{\value{page}}` doesn't work, because the current value of the `page` counter could be 42 even though the text now being processed will wind up on page 43 (see Section 8.1). Instead, use `\label{key}` and `\isodd{\pageref{key}}` (Section C.11.2). The `page` counter does have the expected value when processing the page's head or foot.

`\boolean{nam}` Evaluates to the current value of the boolean register *nam*, where *nam* can be any sequence of letters. This register must be defined with the command `\newboolean{name}`. Its value is set by the command `\setboolean{nam}{bool}`, where *bool* is either `true` or `false`.

**complex expressions** A *test* can be built up from simpler expressions in the customary fashion using the boolean operators `\and`, `\or`, and `\not`, with `\(` and `\)` serving as parentheses.

---

```
\whiledo{test}{body}
```

Repeatedly processes *body* until *test* becomes false, where *test* is the same as for `\ifthenelse`. (Does nothing if *test* is initially false.)

## C.9 Figures and Other Floating Bodies

### C.9.1 Figures and Tables

```
\begin{figure}[loc] body \end{figure}
\begin{figure*}[loc] body \end{figure*}
\begin{table}[loc] body \end{table}
\begin{table*}[loc] body \end{table*}
```

These environments produce floating figures and tables. In two-column format, the ordinary forms produce single-column figures and tables and the *\*-forms produce double-column ones. The two forms are equivalent in single-column format.*

The *body* is typeset in a parbox of width `\textwidth`. It may contain one or more `\caption` commands (see below). The *loc* argument contains a sequence of one to four letters, each one specifying a location where the figure or table may be placed, as follows:

- h** *Here*: at the position in the text where the environment appears. (Not possible for double-column figures and tables in two-column format.)
- t** *Top*: at the top of a text page.
- b** *Bottom*: at the bottom of a text page. (Not possible for double-column figures or tables in two-column format.)
- p** *Page of floats*: on a separate page containing no text, only figures and tables.

If the *loc* argument is missing, the default specifier is `tbp`, so the figure or table may be placed at the top or bottom of a text page or on a separate page consisting only of figures and/or tables.

The *loc* argument can also contain the character `!`, which directs L<sup>A</sup>T<sub>E</sub>X to try harder to place the figure or table at the earliest possible place in the document allowed by the rest of the argument. What “trying harder” means is explained below.

You may find that L<sup>A</sup>T<sub>E</sub>X puts a figure or table where you don’t want it. If the figure or table is printed too soon, you can either move it later in the input or use the `\suppressfloats` command described below. If it is printed too late, you can move it earlier in the input or use an optional argument with a `!` character. Occasionally, you will find that nothing seems to work. You may then think you

have discovered a bug in L<sup>A</sup>T<sub>E</sub>X. You almost certainly haven't. No computer program can deduce exactly where you want your figures to go. L<sup>A</sup>T<sub>E</sub>X's figure-placement algorithm was carefully designed to do the best it could. To solve your problem, you will have to understand why L<sup>A</sup>T<sub>E</sub>X puts the figure or table where it does. L<sup>A</sup>T<sub>E</sub>X follows the rules listed below. You will have to read these rules slowly and carefully to understand what L<sup>A</sup>T<sub>E</sub>X is doing. The last rule, which mentions the formatting parameters, is likely to be the key. You will have to read the descriptions of these parameters carefully to understand the rule. There are fifteen parameters, but one of the first seven is probably responsible for your problem.

Here are the rules that determine where a figure or table is put:

- It is printed at the earliest place that does not violate subsequent rules, except that an `h` (here) position takes precedence over a `t` (top) position.
- It will not be printed on an earlier page than the place in the text where the `figure` or `table` environment appears.
- A figure will not be printed before an earlier figure, and a table will not be printed before an earlier table.<sup>2</sup>
- It may appear only at a position allowed by the `loc` argument, or, if that argument is missing, by the default `tbp` specifier.
- Placement of the figure or table cannot produce an overfull page.
- The page constraints determined by the formatting parameters described below are not violated. However, if a `!` appears in the optional argument, then the constraints for text pages are ignored, and only the ones for float pages (expressed by `\floatpagefraction` and `\dblfloatpagefraction`) apply.

The last three rules are suspended when a `\clearpage`, `\cleardoublepage`, or `\end{document}` command occurs, all unprocessed figures and tables being allowed a `p` option and printed at that point.

When giving an optional `loc` argument, include enough options so these rules allow the figure or table to go somewhere, otherwise it and all subsequent figures or tables will be saved until the end of the chapter or document, probably causing T<sub>E</sub>X to run out of space.

`\caption[lst_entry]{heading}`

Produces a numbered caption.

---

<sup>2</sup>However, in a two-column page style, a single-column figure can come before an earlier double-column figure, and vice versa.

*lst\_entry* Generates the entry in the list of figures or tables. Such an entry should not contain more than a few hundred characters. If this argument is missing, the *heading* argument is used. It is a moving argument.

*heading* The text of the caption. It produces the list of figures or tables entry if the *lst\_entry* argument is missing, in which case it is a moving argument. If this argument contains more than a few hundred characters, a shorter *lst\_entry* argument should be used—even if no list of figures or tables is being produced.

A `\label` command that refers to the caption's number must go in *heading* or after the `\caption` command in the *body* of the `figure` or `table` environment. The `\caption` command can be used only in paragraph mode, but can be placed in a parbox made with a `\parbox` command or `minipage` environment (Section 6.4.3). It is fragile.

### `\suppressfloats [loc]`

Prevents additional figures and tables from appearing on the current page. There are two possible *loc* arguments:

- t** No more figures or tables at the top of the current page.
- b** No more figures or tables at the bottom of the current page.

With no optional argument, additional figures and tables are suppressed from both the top and bottom of the current page. A `!` in the optional argument of a `figure` or `table` environment counteracts the effect of a `\suppressfloats` command for that particular figure or table.

## Style Parameters

Changes made to the following parameters in the preamble apply from the first page on. Changes made afterwards take effect on the next page, not the current one. A *float* denotes either a figure or a table, and a *float page* is a page containing only floats and no text. Parameters that apply to all floats in a one-column page style apply to single-column floats in a two-column style.

**topnumber** A counter whose value is the maximum number of floats allowed at the top of a text page.

**\topfraction** The maximum fraction of the page that can be occupied by floats at the top of the page. Thus, the value `.25` specifies that as much as the top quarter of the page may be devoted to floats. It is changed with `\renewcommand`.

**bottomnumber** Same as `topnumber` except for the bottom of the page.

**\bottomfraction** Same as **\topfraction** except for the bottom of the page.

**totalnumber** A counter whose value is the maximum number of floats that can appear on a single text page, irrespective of their positions.

**\textfraction** The minimum fraction of a text page that must be devoted to text. The other  $1 - \textfraction$  fraction may be occupied by floats. It is changed with **\renewcommand**.

**\floatpagefraction** The minimum fraction of a float page that must be occupied by floats, limiting the amount of blank space allowed on a float page. It is changed with **\renewcommand**.

**dbltopnumber** The analog of **topnumber** for double-column floats on a two-column page.

**\dbltopfraction** The analog of **\topfraction** for double-column floats on a two-column page.

**\dblfloatpagefraction** The analog of **\floatpagefraction** for a float page of double-column floats.

**\floatsep** The vertical space added between floats that appear at the top or bottom of a text page. It is a rubber length.

**\textfloatsep** The vertical space added between the floats appearing at the top or bottom of a page and the text on that page. It is a rubber length.

**\intextsep** The vertical space placed above and below a float that is put in the middle of the text with the **h** location option. It is a rubber length.

**\dblfloatsep** The analog of **\floatsep** for double-width floats on a two-column page. It is a rubber length.

**\dbltextfloatsep** The analog of **\textfloatsep** for double-width floats on a two-column page. It is a rubber length.

### C.9.2 Marginal Notes

**\marginpar** [*left\_text*] {*right\_text*}

Produces a marginal note using *right\_text* if it goes in the right margin or there is no optional argument, otherwise using *left\_text*. The text is typeset in a **parbox**.

For two-sided, single-column printing, the default placement of marginal notes is on the outside margin—left for even-numbered pages, right for odd-numbered ones. For one-sided, single-column printing, the default placement is in the right margin. These defaults may be changed by the following declarations:

`\reversemarginpar` Causes marginal notes to be placed in the opposite margin from the default one.

`\normalmarginpar` Causes marginal notes to be placed in the default margin.

When a marginal note appears within a paragraph, its placement is determined by the declaration in effect at the blank line ending the paragraph. For two-column format, marginal notes always appear in the margin next to the column containing the note, irrespective of these declarations.

A marginal note is normally positioned so its top line is level with the line of text containing the `\marginpar` command; if the command comes between paragraphs, the note is usually level with the last line of the preceding paragraph. However, the note is moved down and a warning message printed on the terminal if this would make it overlap a previous note. Switching back and forth between reverse and normal positioning with `\reversemarginpar` and `\normalmarginpar` may inhibit this movement of marginal notes, resulting in one being overprinted on top of another.

### Style Parameters

`\marginparwidth` The width of the parbox containing a marginal note.

`\marginparsep` The horizontal space between the outer margin and a marginal note.

`\marginparpush` The minimum vertical space allowed between two successive marginal notes.

## C.10 Lining It Up in Columns

### C.10.1 The tabbing Environment

`\begin{tabbing} rows \end{tabbing}`

This environment may be used only in paragraph mode. It produces a sequence of lines, each processed in LR mode, with alignment in columns based upon a sequence of tab stops. Tab stops are numbered 0, 1, 2, etc. Tab stop number  $i$  is said to be *set* if it is assigned a horizontal position on the page. Tab stop 0 is always set to the prevailing left margin (the left margin in effect at the beginning of the environment). If tab stop  $i$  is set, then all tab stops numbered 0 through  $i - 1$  are also set. Tab stop number  $i - 1$  is normally positioned to the left of tab stop number  $i$ .

The behavior of the tabbing commands is described in terms of the values of two quantities called *next.tab.stop* and *left.margin.tab*. Initially, the value of *next.tab.stop* is 1, the value of *left.margin.tab* is 0, and only tab number 0 is set. The value of *next.tab.stop* is incremented by the `\>` and `\=` commands, and

```

Gnat: swatted by: men
 cows
 and gnus
 not very filling
Armadillo: not edible
(note also the: aardvark
 albatross
Gnu: eaten by gnats
 eton)

\begin{tabbing}
 Armadillo: \= \kill
 Gnat: \> swatted by: \= men \+\+ \\
 cows \\
 and \` gnus \- \\
 not very filling \- \\
 Armadillo: \> not edible \\
 \pushtabs
 (note also the: \= aardvark \\
 \> albatross \` eton) \\
 \poptabs
 Gnu: \> eaten by \> gnats
\end{tabbing}

```

Figure C.6: A `tabbing` environment example.

it is reset to the value of `left_margin_tab` by the `\\\` and `\kill` commands. The following commands, all of which are fragile, may appear in `rows`; their use is illustrated in Figure C.6.

- `\=` If the value of `next_tab_stop` is *i*, then this command sets tab stop number *i*'s position to be the current position on the line and changes the value of `next_tab_stop` to *i* + 1.
- `\>` If the value of `next_tab_stop` is *i*, then this command starts the following text at tab stop *i*'s position and changes the value of `next_tab_stop` to *i* + 1.
- `\\\` Starts a new line and sets the value of `next_tab_stop` equal to the value of `left_margin_tab`. See Section C.1.6 for more details.
- `\kill` Throws away the current line, keeping the effects of any tab-stop-setting commands, starts a new line, and sets the value of `next_tab_stop` to the value of `left_margin_tab`.
- `\+` Increases the value of `left_margin_tab` by one. This causes the left margin of subsequent lines to be indented one tab stop to the right, just as if a `\>` command were added to the beginning of subsequent lines. Multiple `\+` commands have the expected cumulative effect.
- `\-` Decreases the value of `left_margin_tab`, which must be positive, by one. This has the effect of canceling one preceding `\+` command, starting with the following line.
- `\<` Decreases the value of `next_tab_stop` by one. This command can be used only at the beginning of a line, where it acts to cancel the effect, on that line, of one previous `\+` command.

- \` Used to put text flush right against the right edge of a column or against the left margin. If the value of *next\_tab\_stop* is *i*, then it causes everything in the current column—all text from the most recent \>, \=, \', \\, or \kill command—to be positioned flush right a distance of \tabbingsep (a style parameter) from the position of tab stop number *i* – 1. Text following the \` command is placed starting at the position of tab stop number *i* – 1.
- \` Moves all following text on the line flush against the prevailing right margin. There must be no \>, \=, or \` command after the \` and before the command that ends the output line.

\pushtabs Saves the current positions of all tab stops, to be restored by a subsequent \poptabs command. You can nest \pushtabs commands, but \pushtabs and \poptabs commands must come in matching pairs within a tabbing environment.

\poptabs See \pushtabs.

\a... The commands \=, \', and \` usually produce accents, but are redefined to tabbing commands inside the **tabbing** environment. The commands \a=, \a', and \a` produce those accents in a **tabbing** environment.

The **tabbing** environment exhibits the following anomalies:

- The scope of a declaration appearing in *rows* is ended by any of the following commands:

|    |    |    |    |           |               |
|----|----|----|----|-----------|---------------|
| \= | \> | \+ | \' | \pushtabs | \kill         |
| \\ | \< | \- | \` | \poptabs  | \end{tabbing} |

No environment contained within the **tabbing** environment can contain any of these tabbing commands.

- The commands \=, \', \`, and \- are redefined to have special meanings inside a **tabbing** environment. The ordinary \- command would be useless in this environment; the effects of the other three are obtained with the \a... command described above. These commands revert to their ordinary meanings inside a parbox contained within the **tabbing** environment.
- One **tabbing** environment cannot be nested within another, even if the inner one is inside a parbox.

### Style Parameters

\tabbingsep See the description of the \` command above.

### C.10.2 The array and tabular Environments

```
\begin{array}[pos]{cols} rows \end{array}
\begin{tabular}[pos]{cols} rows \end{tabular}
\begin{tabular*}[width]{pos}{cols} rows \end{tabular*}
```

These environments produce a box (Section 6.4.3) consisting of a sequence of rows of items, aligned vertically in columns. The `array` environment can be used only in math mode, while `tabular` and `tabular*` can be used in any mode. Examples illustrating most of the features of these environments appear in Figure C.7.

*wdth* Specifies the width of the `tabular*` environment. There must be rubber space between columns that can stretch to fill out the specified width; see the `\extracolsep` command below.

*pos* Specifies the vertical positioning; the default is alignment on the center of the environment.

**t** align on top row.

**b** align on bottom row.

| GG&A Hoofed Stock |          |      |                                       |
|-------------------|----------|------|---------------------------------------|
| Year              | Price    |      | Comments                              |
|                   | low      | high |                                       |
| 1971              | 97-245   |      | Bad year.                             |
| 72                | 245-245  |      | Light trading due to a heavy winter.  |
| 73                | 245-2001 |      | No gnus was very good gnus this year. |

←———— 65mm —————→

Table

|       |         |                 |
|-------|---------|-----------------|
| 1.234 | centaur | rite::gauche    |
| 56.7  | scenter | wright::rad     |
| 8.99  | cent    | write::sinister |

```
\begin{tabular}{|r||r@{-->}l|p{1.25in}|}
\hline
\multicolumn{4}{|c|}{GG\&A Hoofed Stock}
&\hline\hline
&\multicolumn{2}{c|}{Price}& \multicolumn{2}{c}{Comments}\\
&\multicolumn{1}{c|}{Year}&&\multicolumn{1}{c}{low}& high &\multicolumn{1}{c}{Comments}\\
&&&\hline
1971 & 97 & 245 & Bad year. &\hline
72 & 245 & 245 & Light trading due to a heavy winter. &\hline
73 & 245 & 2001 & No gnus was very good gnus this year. &\hline
\end{tabular}

\begin{tabular*}{65mm}{@{}r@{.}l@{}}
@{\extracolsep{\fill}}cr%
@{\extracolsep{0pt}::}l@{}}
\multicolumn{5}{c}{\underline{Table}} \\
1&234 & centaur & rite & gauche \\
56 & 7 & scenter & wright & rad \\
8&99 & cent & write & sinister
\end{tabular*}
```

Figure C.7: Examples of the `tabular` and `tabular*` environments.

*cols* Specifies the column formatting. It consists of a sequence of the following specifiers, corresponding to the sequence of columns and intercolumn material:

- l A column of left-aligned items.
- r A column of right-aligned items.
- c A column of centered items.
- | A vertical line the full height and depth of the environment.

**$\@{text}$**  This specifier is called an  *$\@$ -expression*. It inserts *text* in every row, where *text* is processed in math mode in the `array` environment and in LR mode in the `tabular` and `tabular*` environments. The *text* is considered a moving argument, so any fragile command within it must be `\protect`'ed.

An  *$\@$ -expression* suppresses the space L<sup>A</sup>T<sub>E</sub>X normally inserts between columns; any desired space between the inserted text and the adjacent items must be included in *text*. To change the space between two columns from the default to *wd*, put an  **$\@{\hspace{wd}}$**  command (Section 6.4.1) between the corresponding column specifiers.

An  **$\@extracolsep{wd}$**  command in an  *$\@$ -expression* causes an extra space of width *wd* to appear to the left of all subsequent columns, until countermanded by another  **$\@extracolsep$**  command. (However, it will not put space to the left of the first column.) Unlike ordinary intercolumn space, this extra space is not suppressed by an  *$\@$ -expression*. An  **$\@extracolsep$**  command can be used only in an  *$\@$ -expression* in the *cols* argument. It is most commonly used to insert a  **$\@fill$**  space (Section 6.4.1) in a `tabular*` environment.

**$\@p{wd}$**  Produces a column with each item typeset in a `parbox` of width *wd*, as if it were the argument of a  **$\@parbox[t]{wd}$**  command (Section 6.4.3). However, a  **$\@\\$**  may not appear in the item, except in the following situations: (i) inside an environment like `minipage`, `array`, or `tabular`, (ii) inside an explicit `parbox`, or (iii) in the scope of a `\centering`, `\raggedright`, or `\raggedleft` declaration. The latter declarations must appear inside braces or an environment when used in a *p*-column element.

**$\@*{num}{cols}$**  Equivalent to *num* copies of *cols*, where *num* is any positive integer and *cols* is any list of column-specifiers, which may contain another  *$\@$ -expression*.

An extra space, equal to half the default intercolumn space, is put before the first column unless *cols* begins with a  **$\@|$**  or  *$\@$ -expression*, and after the last column unless *cols* ends with a  **$\@|$**  or  *$\@$ -expression*. This space

usually causes no problem, but is easily eliminated by putting an `\{\}` at the beginning and end of *cols*.

*rows* A sequence of rows separated by `\\"` commands (Section C.1.6). Each row is a sequence of items separated by `\&` characters; it should contain the same number of items as specified by the *cols* argument. Each item is processed as if it were enclosed in braces, so the scope of any declaration in an item lies within that item. The following commands may appear in an item:

`\multicolumn{num}{col}{item}` Makes *item* the text of a single item spanning *num* columns, positioned as specified by *col*. If *num* is 1, then the command serves simply to override the item positioning specified by the environment argument. The *col* argument must contain exactly one `l`, `r`, or `c` and may contain one or more `\{`-expressions and `|` characters. It replaces that part of the environment's *cols* argument corresponding to the *num* spanned columns, where the part corresponding to any column except the first begins with `l`, `r`, `c`, or `p`, so the *cols* argument `|c|l\{\cdot\}lr` has the four parts `|c|`, `\{\cdot\}`, `l`, and `r`. A `\multicolumn` command must either begin the row or else immediately follow an `\&`. It is fragile.

`\vline` When used within an `l`, `r`, or `c` item, it produces a vertical line extending the full height and depth of its row. An `\hfill` command (Section 6.4.2) can be used to move the line to the edge of the column. A `\vline` command can also be used in an `\{`-expression. It is robust.

The following commands can go between rows to produce horizontal lines. They must appear either before the first row or immediately after a `\\"` command. A horizontal line after the last row is produced by ending the row with a `\\"` followed by one of these commands. (This is the only case in which a `\\"` command appears after the last row of an environment.) These commands are fragile.

`\hline` Draws a horizontal line extending the full width of the environment. Two `\hline` commands in succession leave a space between the lines; vertical rules produced by `|` characters in the *cols* argument do not appear in this space.

`\cline{col1-col2}` Draws a horizontal line across columns *col<sub>1</sub>* through *col<sub>2</sub>*. Two or more successive `\cline` commands draw their lines in the same vertical position. See the `\multicolumn` command above for how to determine what constitutes a column.

The following properties of these environments, although mentioned above, are often forgotten:

- These environments make a box; see Section 6.5 for environments and commands that can be used to position this box.
- The box made by these commands may have blank space before the first column and after the last column; this space can be removed with an `\@-expression`.
- Any declaration in `rows` is within an item; its scope is contained within the item.
- An `\@-expression` in `cols` suppresses the default intercolumn space.

## Style Parameters

The following style parameters can be changed anywhere outside an `array` or `tabular` environment. They can also be changed locally within an item, but the scope of the change should be explicitly delimited by braces or an environment.

`\arraycolsep` Half the width of the default horizontal space between columns in an `array` environment.

`\tabcolsep` Half the width of the default horizontal space between columns in a `tabular` or `tabular*` environment.

`\arrayrulewidth` The width of the line created by a `|` in the `cols` argument or by an `\hline`, `\cline`, or `\vline` command.

`\doublerulesep` The width of the space between lines created by two successive `|` characters in the `cols` argument, or by two successive `\hline` commands.

`\arraystretch` Controls the spacing between rows. The normal interrow space is multiplied by `\arraystretch`, so changing it from its default value of 1 to 1.5 makes the rows 1.5 times farther apart. Its value is changed with `\renewcommand` (Section 3.4).

## C.11 Moving Information Around

### C.11.1 Files

`LATEX` creates a number of ancillary files when processing a document. They all have the same first name as the root file (Section 4.4). These files are referred to, and listed below, by their extensions. A `\nofiles` command in the preamble prevents `LATEX` from writing any of them except the `dvi` and `log` files. Knowing when and under what circumstances these files are read and written can help in locating and recovering from errors.

**aux** Used for cross-referencing and in compiling the table of contents, list of figures, and list of tables. In addition to the main **aux** file, a separate **aux** file is also written for each `\include`'d file (Section 4.4), having the same first name as that file. All **aux** files are read by the `\begin{document}` command. The `\begin{document}` command also starts writing the main **aux** file; writing of an `\include`'d file's **aux** file is begun by the `\include` command and is ended when the `\include`'d file has been completely processed. A `\nofiles` command suppresses the writing of all **aux** files.

The table of contents and cross-reference information in the **aux** files can be printed by running **L<sup>A</sup>T<sub>E</sub>X** on the file `lablst.tex`.

- bb1** This file is written by **BIB<sub>E</sub>X**, not by **L<sup>A</sup>T<sub>E</sub>X**, using information on the **aux** file. It is read by the `\bibliography` command.
- dvi** This file contains **L<sup>A</sup>T<sub>E</sub>X**'s output, in a form that is independent of any particular printer. (This printer-independence may be lost when using the **graphics**, **color**, and **pict2e** packages; see the introduction to Chapter 7.) Another program must be run to print the information on the **dvi** file. The file is always written unless **L<sup>A</sup>T<sub>E</sub>X** has generated no printed output.
- glo** Contains the `\glossaryentry` commands generated by `\glossary` commands. The file is written only if there is a `\makeglossary` command and no `\nofiles` command.
- idx** Contains the `\indexentry` commands generated by `\index` commands. The file is written only if there is a `\makeindex` command and no `\nofiles` command.
- ind** This file is written by *MakeIndex*, not by **L<sup>A</sup>T<sub>E</sub>X**, using information on the **idx** file. It is read by the `\printindex` command. See Appendix A.
- lof** Read by the `\listoffigures` command to generate a list of figures; it contains the entries generated by all `\caption` commands in **figure** environments. The **lof** file is generated by the `\end{document}` command. It is written only if there is a `\listoffigures` command and no `\nofiles` command.
- log** Contains everything printed on the terminal when **L<sup>A</sup>T<sub>E</sub>X** is executed, plus additional information and some extra blank lines. It is always written. In some systems, this file has an extension other than **log**.
- lot** Read by the `\listoftables` command to generate a list of tables; it contains the entries generated by all `\caption` commands in **table** environments. The **lot** file is generated by the `\end{document}` command. It is written only if there is a `\listoftables` command and no `\nofiles` command.

**toc** Read by the `\tableofcontents` command to generate a table of contents; it contains the entries generated by all sectioning commands (except the `*-forms`). The `toc` file is generated by the `\end{document}` command. It is written only if there is a `\tableofcontents` command and no `\nofiles` command.

## C.11.2 Cross-References

```
\label {key}
\ref {key}
\pageref{key}
```

The `key` argument is any sequence of letters, digits, and punctuation symbols; upper- and lowercase letters are regarded as different. L<sup>A</sup>T<sub>E</sub>X maintains a *current \ref value*, which is set with the `\refstepcounter` declaration (Section C.8.4). (This declaration is issued by the sectioning commands, by numbered environments like `equation`, and by an `\item` command in an `enumerate` environment.) The `\label` command writes an entry on the `aux` file (Section C.11.1) containing `key`, the current `\ref` value, and the number of the current page. When this `aux` file entry is read by the `\begin{document}` command (the next time L<sup>A</sup>T<sub>E</sub>X is run on the same input file), the `\ref` value and page number are associated with `key`, causing a `\ref{key}` or `\pageref{key}` command to produce the associated `\ref` value or page number, respectively.

The `\label` command is fragile, but it can be used in the argument of a sectioning or `\caption` command.

## C.11.3 Bibliography and Citation

```
\bibliography{bib_files}
```

Used in conjunction with the BIBT<sub>E</sub>X program (Section 4.3.1) to produce a bibliography. The `bib_files` argument is a list of first names of bibliographic database (`bib`) files, separated by commas; these files must have the extension `bib`. The `\bibliography` command does two things: (i) it creates an entry on the `aux` file (Section C.11.1) containing `bib_files` that is read by BIBT<sub>E</sub>X, and (ii) it reads the `bbl` file (Section C.11.1) generated by BIBT<sub>E</sub>X to produce the bibliography. (The `bbl` file will contain a `thebibliography` environment.) The database files are used by BIBT<sub>E</sub>X to create the `bbl` file.

```
\begin{thebibliography}{widest_label} entries \end{thebibliography}
```

Produces a bibliography or source list. In the standard `article` document class, this source list is labeled “References”; in the `report` and `book` class, it is labeled “Bibliography”. See Section 6.1.4 for information on how to create a document-class option to change the reference list’s label.

*widest\_label* Text that, when printed, is approximately as wide as the widest item label produced by the `\bibitem` commands in *entries*. It controls the formatting.

*entries* A list of entries, each begun by the command

`\bibitem[label]{cite_key}`

which generates an entry labeled by *label*. If the *label* argument is missing, a number is generated as the label, using the `enumiv` counter. The *cite\_key* is any sequence of letters, numbers, and punctuation symbols not containing a comma. This command writes an entry on the `aux` file (Section C.11.1) containing *cite\_key* and the item's label. When this `aux` file entry is read by the `\begin{document}` command (the next time L<sup>A</sup>T<sub>E</sub>X is run on the same input file), the item's label is associated with *cite\_key*, causing a reference to *cite\_key* by a `\cite` command to produce the associated label.

`\cite[text]{key_list}`

The *key\_list* argument is a list of citation keys (see `\bibitem` above). This command generates an in-text citation to the references associated with the keys in *key\_list* by entries on the `aux` file read by the `\begin{document}` command. It also writes *key\_list* on the `aux` file, causing BIB<sub>T</sub>E<sub>X</sub> to add the associated references to the bibliography (Section 4.3.1). If present, *text* is added as a remark to the citation. Fragile.

`\nocite{key_list}`

Produces no text, but writes *key\_list*, which is a list of one or more citation keys, on the `aux` file. This causes BIB<sub>T</sub>E<sub>X</sub> to add the associated references to the bibliography (Section 4.3.1). A `\nocite{*}` command causes BIB<sub>T</sub>E<sub>X</sub> to add all references from the `bib` files. The `\nocite` command must appear after the `\begin{document}`. It is fragile.

#### C.11.4 Splitting the Input

`\input{file_name}`

Causes the indicated file to be read and processed, exactly as if its contents had been inserted in the current file at that point. The *file\_name* may be a complete file name with extension or just a first name, in which case the file *file\_name.tex* is used. If the file cannot be found, an error occurs and L<sup>A</sup>T<sub>E</sub>X requests another file name.

---

```
\include{file}
\includeonly{file_list}
```

Used for the selective inclusion of files. The *file* argument is the first name of a file, denoting the file *file.tex*, and *file\_list* is a possibly empty list of first names of files separated by commas. If *file* is one of the file names in *file\_list* or if there is no *\includeonly* command, then the *\include* command is equivalent to

```
\clearpage \input{file} \clearpage
```

except that if file *file.tex* does not exist, then a warning message rather than an error is produced. If *file* is not in *file\_list*, the *\include* command is equivalent to *\clearpage*.

The *\includeonly* command may appear only in the preamble; an *\include* command may not appear in the preamble or in a file read by another *\include* command. Both commands are fragile.

```
\begin{filecontents}{nam} body \end{filecontents}
\begin{filecontents*}{nam} body \end{filecontents*}
```

If a file named *nam* does not exist, then one is created having *body* as its contents. If file *nam* already exists, then a warning message is printed and no file is written. The *filecontents* environment writes helpful identifying comments at the beginning of the file. These comments can cause problems if the file is used as input to a program that, unlike L<sup>A</sup>T<sub>E</sub>X, does not treat lines beginning with % as comments. The *filecontents\** environment does not add any comments. These environments can appear only before the *\documentclass* command.

```
\listfiles
```

Causes L<sup>A</sup>T<sub>E</sub>X to print on the terminal a list of all files that it reads when processing the document, excluding ancillary files that it wrote. The command may appear only in the preamble.

## C.11.5 Index and Glossary

Appendix A describes how to make an index using the *MakeIndex* program.

### Producing an Index

```
\begin{theindex} text \end{theindex}
```

Produces a double-column index. Each entry is begun with either an *\item* command, a *\subitem* command, or a *\subsubitem* command.

```
\printindex
```

Defined by the *makeidx* package. This command just reads the *ind* file.

### Compiling the Entries

**\makeindex** Causes the `\indexentry` entries produced by `\index` commands to be written on the `idx` file, unless a `\nofiles` declaration occurs. The `\makeindex` command may appear only in the preamble.

**\makeglossary** Causes the `\glossaryentry` entries produced by `\glossary` commands to be written on the `glo` file, unless a `\nofiles` declaration occurs. The `\makeglossary` command may appear only in the preamble.

**\index{*str*}** If an `idx` file is being written, then this command writes an `\indexentry{str}{pg}` entry on it, where *pg* is the page number. The *str* argument may contain any characters, including special characters, but it must have no unmatched braces, where the braces in `\{` and `\}` are included in the brace matching. The `\index` command may not appear inside another command's argument unless *str* contains only letters, digits, and punctuation characters. The command is fragile.

**\glossary{*str*}** If a `glo` file is being written, then this command writes a `\glossaryentry{str}{pg}` entry on it, where *str* and *pg* are the same as in the `\index` command, described above. The `\glossary` command may not appear inside another command's argument unless *str* contains only letters, digits, and punctuation characters. The command is fragile.

### C.11.6 Terminal Input and Output

**\typeout{*msg*}**

Prints *msg* on the terminal and in the log file. Commands in *msg* that are defined with `\newcommand` or `\renewcommand` are replaced by their definitions before being printed.  $\text{\LaTeX}$  commands in *msg* may produce strange results. Preceding a command name by `\protect` causes that command name to be printed.

$\text{\TeX}$ 's usual rules for treating multiple spaces as a single space and ignoring spaces after a command name apply to *msg*. A `\space` command in *msg* causes a single space to be printed. The `\typeout` command is fragile; moreover, putting it in the argument of another  $\text{\TeX}$  command may do strange things. The *msg* argument is a moving argument.

**\typein[*cmd*]{*msg*}**

Prints *msg* on the terminal, just like `\typeout{msg}`, and causes  $\text{\TeX}$  to stop and wait for you to type a line of input, ending with *return*. If the *cmd* argument is missing, the typed input is processed as if it had been included in the input file in place of the `\typein` command. If the *cmd* argument is present, it must be a command name. This command name is then defined or redefined to be

the typed input. Thus, if *cmd* is not already defined, then the command acts like

```
\typeout{msg}
\newcommand{cmd}{typed input}
```

The `\typein` command is fragile; moreover, it may produce an error if it appears in the argument of a L<sup>A</sup>T<sub>E</sub>X command. The *msg* argument is a moving argument.

## C.12 Line and Page Breaking

### C.12.1 Line Breaking

```
\linebreak [num]
\nolinebreak [num]
```

The `\linebreak` command encourages and `\nolinebreak` discourages a line break, by an amount depending upon *num*, which is a digit from 0 through 4. A larger value of *num* more strongly encourages or discourages the line break; the default is equivalent to a *num* argument of 4, which either forces or completely prevents a line break. An underfull `\hbox` message is produced if a `\linebreak` command results in too much space between words on the line. Both commands are fragile.

```
\ \
*[len]
\newline
```

These commands start a new line without justifying the current one, producing a ragged-right effect. The optional argument of `\` adds an extra vertical space of length *len* above the new line. The `*-form` inhibits a page break right before the new line. The `\newline` command may be used only in paragraph mode and should appear within a paragraph; it produces an underfull `\hbox` warning and extra vertical space if used at the end of a paragraph, and an error when used between paragraphs. The `\` command behaves the same way when used in paragraph mode. Both commands are fragile.

`\-`

Permits the line to be hyphenated (the line broken and a hyphen inserted) at that point. It inhibits hyphenation at any other point in the current word except where allowed by another `\-` command. Robust.

```
\hyphenation{words}
```

Declares allowed hyphenation points, where *words* is a list of words, separated by spaces, in which each hyphenation point is indicated by a `-` character. It is a global declaration (Section C.1.4) and is robust.

```
\sloppy
\fussy
```

Declarations that control line breaking. The `\fussy` declaration, which is the default, prevents too much space between words, but leaves words extending past the right-hand margin if no good line break is found. The `\sloppy` declaration almost always breaks lines at the right-hand margin, but may leave too much space between words, in which case TeX produces an underfull `\hbox` warning. Line breaking is controlled by the declaration in effect at the blank line or `\par` command that ends the paragraph.

```
\begin{sloppypar} pars \end{sloppypar}
```

Typesets *pars*, which must consist of one or more complete paragraphs, with the `\sloppy` declaration in effect.

### C.12.2 Page Breaking

```
\pagebreak [num]
\nopagebreak [num]
```

The `\pagebreak` command encourages and `\nopagebreak` discourages column breaking by an amount depending upon *num*, where the entire page is a single column in a one-column page style. The *num* argument is a digit from 0 through 4, a larger value more strongly encouraging or discouraging a break; the default is equivalent to *num* having the value 4, which forces or entirely forbids a break. When used within a paragraph, these commands apply to the point immediately following the line in which they appear. When `\flushbottom` is in effect (Section 6.1.1), an underfull `\vbox` message is produced if `\pagebreak` results in too little text on the page. A `\nopagebreak` command will have no effect if another TeX command has explicitly allowed a page break to occur at that point. These commands have no effect when used in LR mode or inside a box. Both commands are fragile.

```
\enlargethispage {len}
\enlargethispage*{len}
```

These commands increase the height of the page that TeX is currently trying to produce by *len*, which must be a rigid length and may be negative. The `*-form` shrinks the vertical space on the page as much as possible, which is what you want to do when trying to squeeze a little more onto a page than TeX wants to put there. For two-sided printing, it is usually best to make facing pages the same height. These commands are fragile.

```
\newpage
\clearpage
\cleardoublepage
```

When one-column pages are being produced, these commands all end the current paragraph and the current page. Any unfilled space in the body of the page (Section 6.1.2) appears at the bottom, even with `\flushbottom` in effect (Section 6.1.1). The `\clearpage` and `\cleardoublepage` commands also cause all figures and tables that have so far appeared in the input to be printed, using one or more pages of only figures and/or tables if necessary. In a two-sided printing style, `\cleardoublepage` also makes the next page a right-hand (odd-numbered) page, producing a blank page if necessary.

When two-column text is being produced, `\newpage` ends the current column rather than the current page; `\clearpage` and `\cleardoublepage` end the page, producing a blank right-hand column if necessary. These commands should be used only in paragraph mode; they should not be used inside a parbox (Section 6.4.3). The `\newpage` and `\clearpage` commands are robust; `\cleardoublepage` is fragile.

## C.13 Lengths, Spaces, and Boxes

### C.13.1 Length

**explicit lengths** An explicit length is written as an optional sign (+ or -) followed by a decimal number (a string of digits with an optional decimal point) followed by a *dimensional unit*. The following dimensional units are recognized by TeX:

**cm** Centimeters.

**em** One em is about the width of the letter *M* in the current font.

**ex** One ex is about the height of the letter *x* in the current font.

**in** Inches.

**pc** Picas (1pc = 12pt).

**pt** Points (1in = 72.27pt).

**mm** Millimeters.

**\fill** A rubber length (Section 6.4.1) having a natural length of zero and the ability to stretch to any arbitrary (positive) length. Robust.

**\stretch{dec\_num}** A rubber length having zero natural length and *dec\_num* times the stretchability of `\fill`, where *dec\_num* is a signed decimal number (an optional sign followed by a string of digits with an optional decimal point). Robust.

`\newlength{cmd}` Declares *cmd* to be a length command, where *cmd* is the name of a command not already defined. The value of *cmd* is initialized to zero inches. Fragile.

`\setlength{cmd}{len}` Sets the value of the length command *cmd* equal to *len*. Robust.

`\addtolength{cmd}{len}` Sets the value of the length command *cmd* equal to its current value plus *len*. Robust.

`\settowidth {cmd}{text}`  
`\settoheight{cmd}{text}`  
`\settodepth {cmd}{text}` Set the value of the length command *cmd* equal to the natural width, height, and depth, respectively, of the output generated when *text* is typeset in LR mode. Robust.

### C.13.2 Space

`\hspace {len}`  
`\hspace*{len}`

Produce a horizontal space of width *len*. The space produced by `\hspace` is removed if it falls at a line break; that produced by `\hspace*` is not. These commands are robust.

`\vspace {len}`  
`\vspace*{len}`

Add a vertical space of height *len*. If the command appears in the middle of a paragraph, then the space is added after the line containing it. The space produced by `\vspace` is removed if it falls at a page break; that produced by `\vspace*` is not. These commands may be used only in paragraph mode; they are fragile.

`\bigskip`  
`\medskip`  
`\smallskip`

These commands are equivalent to the three commands

|                                      |                                        |
|--------------------------------------|----------------------------------------|
| <code>\vspace{\bigskipamount}</code> | <code>\vspace{\smallskipamount}</code> |
| <code>\vspace{\medskipamount}</code> |                                        |

where the three length commands `\bigskipamount`, `\medskipamount`, and `\smallskipamount` are style parameters. These space-producing commands can be used in the definitions of environments to provide standard amounts of vertical space. They are fragile.

---

**\addvspace{*len*}**

This command normally adds a vertical space of height *len*. However, if vertical space has already been added to the same point in the output by a previous **\addvspace** command, then this command will not add more space than needed to make the natural length of the total vertical space equal to *len*. It is used to add the extra vertical space above and below most L<sup>A</sup>T<sub>E</sub>X environments that start a new paragraph. It may be used only in paragraph mode between paragraphs—that is, after a blank line or **\par** command (in T<sub>E</sub>X's vertical mode). *Fragile*.

**\hfill**

Equivalent to **\hspace{\fill}**.

**\vfill**

Equivalent to a blank line followed by **\vspace{\fill}**; it should be used only in paragraph mode.

### C.13.3 Boxes

A *box* is an object that is treated by T<sub>E</sub>X as a single character, so it will not be broken across lines or pages.

**\mbox{*text*}**  
**\makebox[*wdth*][*pos*]{*text*}**

Typesets *text* in LR mode in a box. The box has the width of the typeset text except for a **\makebox** command with a *wdth* argument, in which case it has width *wdth*. In the latter case, the position of the text within the box is determined by the one-letter *pos* argument as follows:

- 1 Flush against left edge of box.
- r Flush against right edge of box.
- s Interword space in *text* is stretched or shrunk to try to fill the box exactly.

The default positioning is centered in the box. The **\mbox** command is robust; **\makebox** is fragile.

**\fbox{*text*}**  
**\framebox[*wdth*][*pos*]{*text*}**

Similar to **\mbox** and **\makebox**, except that a rectangular frame is drawn around the resulting box. The **\fbox** command is robust; **\framebox** is fragile.

---

```
\newsavebox{cmd}
```

Declares *cmd*, which must be a command name that is not already defined, to be a bin for saving boxes. Fragile.

```
\sbox{cmd}{text}
\savebox{cmd}[wdth][pos]{text}
\begin{lrbox}{cmd} text \end{lrbox}
```

Typeset *text* in a box just as for `\makebox`. However, instead of printing the resulting box, they save it in bin *cmd*, which must have been declared with `\newsavebox`. In the `lrbox` environment, spaces are removed from the beginning and end of *text*. The `\sbox` command is robust; `\savebox` is fragile.

```
\usebox{cmd}
```

Prints the box most recently saved in bin *cmd*. Robust.

```
\parbox [pos]{wdth}{text}
\begin{minipage}[pos]{wdth} text \end{minipage}
```

They produce a *parbox*—a box of width *wdth* formed by typesetting *text* in paragraph mode. The vertical positioning of the box is specified by the one-letter *pos* argument as follows:

- b** The bottom line of the box is aligned with the current line of text.
- t** The top line of the box is aligned with the current line of text.

The default vertical positioning is to align the center of the box with the center of the current line of text.

The list-making environments listed in Section 6.6 and the `tabular` environment may appear in *text* with the `minipage` environment, but not with the `\parbox` command. (If *text* consists of only a `tabbing` environment, then the width of the resulting box is the actual width of the longest line rather than the *wdth* argument.) A `\footnote` or `\footnotetext` command appearing in *text* in a `minipage` environment produces a footnote at the bottom of the parbox ended by the next `\end{minipage}` command, which may be the wrong place for it when there are nested `minipage` environments. These footnote-making commands may not be used in the *text* argument of `\parbox`.

A `minipage` environment that begins with a displayed equation or with an `eqnarray` or `eqnarray*` environment will have extra vertical space at the top (except with the `fleqn` document-class option). This extra space can be removed by starting *text* with a `\vspace{-\abovedisplayskip}` command.

The `\parbox` command is fragile.

---

```
\rule[raise_len]{wdth}{hght}
```

Generates a solid rectangle of width *wdth* and height *hght*, raised a distance of *raise\_len* above the bottom of the line. (A negative value of *raise\_len* lowers it.) The default value of *raise\_len* is zero inches. Fragile.

```
\raisebox{raise_len}[hght][dpth]{text}
```

Creates a box by typesetting *text* in LR mode, raising it by *raise\_len*, and pretending that the resulting box extends a distance of *hght* above the bottom of the current line and a distance of *dpth* below it. If the *dpth* argument or both optional arguments are omitted, TeX uses the actual extent of the box. Fragile.

```
\width
\height
\depth
\totalheight
```

Length commands that can be used only in the *wdth* argument of `\makebox`, `\framebox`, and `\savebox`, and in the *raise\_len*, *hght*, and *dpth* arguments of `\raisebox`. They refer to the dimensions (width, height, depth, and height + depth) of the box obtained by typesetting the *text* argument.

## Style Parameters

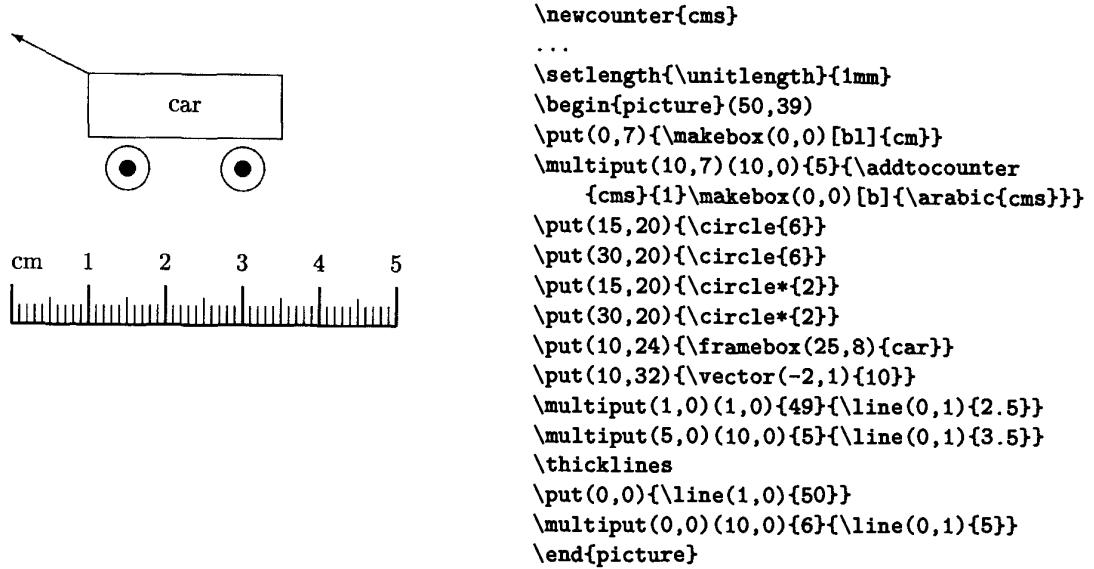
`\fboxrule` The width of the lines forming the box produced by `\fbox` and `\framebox`. However, the version of `\framebox` used in the `picture` environment (Section 7.1) employs the same width lines as other picture commands.

`\fboxsep` The amount of space left between the edge of the box and its contents by `\fbox` and `\framebox`. It does not apply to the version of `\framebox` used in the `picture` environment (Section 7.1).

## C.14 Pictures and Color

### C.14.1 The picture Environment

A *coordinate* is a decimal number—an optional sign followed by a string of digits with an optional decimal point. It represents a length in multiples of `\unitlength`. All argument names in this section that begin with *x* or *y* are coordinates.

Figure C.8: A sample `picture` environment.

```

\begin{picture}(x_dimen,y_dimen)(x_offset,y_offset)
 pict_cmds
\end{picture}

```

Creates a box of width *x\_dimen* and height *y\_dimen*, both of which must be non-negative. The *(x\_offset, y\_offset)* argument is optional. If present, it specifies the coordinates of the lower-left corner of the picture; if absent, the lower-left corner has coordinates (0, 0). (Like all dimensions in the `picture` environment, the lengths specified by the arguments of the `picture` environment are given in multiples of `\unitlength`.) The `picture` environment can be used anywhere that ordinary text can, including within another `picture` environment.

The *pict\_cmds* are processed in picture mode—a special form of LR mode—and may contain only declarations and the following commands:

```

\put \multiput \qbezier \graphpaper

```

Figure C.8 illustrates many of the picture-drawing commands described below.

### Picture-Mode Commands

The following are the only commands, other than declarations, that can be used in picture mode.

---

```
\put(x_coord, y_coord){picture_object}
```

Places *picture\_object* in the picture with its reference point at the position specified by coordinates  $(x\_coord, y\_coord)$ . The *picture\_object* can be arbitrary text, which is typeset in LR mode, or else one of the special picture-object commands described below. The `\put` command is fragile.

```
\multiput(x_coord, y_coord) (x_incr, y_incr) {num}{picture_object}
```

Places *num* copies of *picture\_object*, the *i*<sup>th</sup> one positioned with its reference point having coordinates  $(x\_coord + [i - 1]x\_incr, y\_coord + [i - 1]y\_incr)$ . The *picture\_object* is the same as for the `\put` command above. It is typeset *num* times, so the copies need not be identical if it includes declarations. (See Figure C.8.) Fragile.

```
\qbezier [num] (x_coord1, y_coord1) (x_coord2, y_coord2) (x_coord3, y_coord3)
```

Draws a quadratic Bezier curve whose control points are specified by the mandatory arguments (see Section 7.1.3). The *num* argument specifies the number of points plotted on the curve. If it is absent, a smooth curve is drawn, with the maximum number of points determined by the value of `\qbeziermax`. Use `\renewcommand` to change `\qbeziermax`:

```
\renewcommand{\qbeziermax}{250}
```

(With the `pict2e` package, there is no limit to the number of points plotted.)

```
\graphpaper [spcng] (x_coord, y_coord) (x_dimen, y_dimen)
```

Defined by the `graphpaper` package. It draws a coordinate grid with origin at  $(x\_coord, y\_coord)$ , extending *x\_dimen* units to the right and *y\_dimen* units up. Grid lines are spaced every *spcng* units; the default value is 10. All arguments must be integers.

## Picture Objects

```
\makebox (x_dimen, y_dimen) [pos]{text}
\framebox(x_dimen, y_dimen) [pos]{text}
\dashbox {dash_dimen}(x_dimen, y_dimen) [pos]{text}
```

Produce a box having width *x\_dimen* and height *y\_dimen* (in multiples of `\unitlength`) with reference point at its lower-left corner. The *text* is typeset in LR mode, positioned in the box as specified by the one- or two-letter *pos* argument as follows:

- 1 Horizontally positioned flush against the left edge of the box.
- r Horizontally positioned flush against the right edge of the box.

**t** Vertically positioned flush against the top edge of the box.

**b** Vertically positioned flush against the bottom edge of the box.

The default horizontal and vertical positioning is to center *text* in the box. The `\framebox` command also draws a rectangle showing the edges of the box, and `\dashbox` draws the rectangle with dashed lines, composed of dashes and spaces of length *dash\_dimen* (in multiples of `\unitlength`), where *dash\_dimen* is a positive decimal number. For best results, *x\_dimen* and *y\_dimen* should be integral multiples of *dash\_dimen*. The thickness of the lines drawn by `\framebox` and `\dashbox` equals the width of the lines produced by other picture commands; it is not determined by `\fboxrule`. All three commands are fragile.

```
\line (h_slope,v_slope){dimen}
\vector(h_slope,v_slope){dimen}
```

Draw a line having its reference point at the beginning and its slope determined by  $(h\_slope, v\_slope)$ , where *h\_slope* and *v\_slope* are positive or negative integers of magnitude at most 6 for `\line` and at most 4 for `\vector`, with no common divisors except  $\pm 1$ . (These restrictions are eliminated by the `pict2e` package.) In addition, `\vector` draws an arrowhead at the opposite end of the line from the reference point. The horizontal extent of the line is *dimen* (in multiples of `\unitlength`) unless *h\_slope* is zero, in which case *dimen* is the (vertical) length of the line. However, a line that is neither horizontal nor vertical may not be drawn unless *dimen* times `\unitlength` is at least 10 points (1/7 inch). (This does not apply when the `pict2e` package is loaded.) The `\vector` command always draws the arrowhead. Both commands are fragile.

```
\shortstack[pos]{col}
```

The *pos* argument must be either *l*, *r*, or *c*, the default being equivalent to *c*. This command produces the same result as

```
\begin{tabular}[b]{pos} col \end{tabular}
```

(Section 3.6.2) except that no space is left on either side of the resulting box and there is usually less interrow space. The reference point is at the left edge of the box, level with the reference point of the bottom line. Fragile.

```
\circle {diam}
\circle*{diam}
```

Draw a (hollow) circle and a disk (filled circle), respectively, with diameter as close as possible to *diam* times `\unitlength` and reference point in the center of the circle. The largest circle L<sup>A</sup>T<sub>E</sub>X can draw has a diameter of 40 points (about 1/2 inch) and the largest disk has a diameter of 15 points (about .2 inch). (With the `pict2e` package, any size circle or disk can be drawn.) Both commands are fragile.

```
\oval [rad] (x_dimen, y_dimen) [part]
```

Draws an oval inscribed in a rectangle of width *x\_dimen* and height *y\_dimen*, its corners made with quarter circles of the largest possible radius less than or equal to *rad*. An explicit *rad* argument can be used only with the *pict2e* package; the default value is the radius of the largest quarter-circle L<sup>A</sup>T<sub>E</sub>X can draw without the *pict2e* package. The *part* argument consists of one or two of the following letters to specify a half or quarter oval: **l** (left), **r** (right), **t** (top), **b** (bottom). The default is to draw the entire oval. The reference point is the center of the (complete) oval. Fragile.

```
\frame{picture_object}
```

Puts a rectangular frame around *picture\_object*. The reference point is the bottom left corner of the frame. No extra space is put between the frame and *picture\_object*. Fragile.

## Picture Declarations

The following declarations can appear anywhere in the document, including in picture mode. They obey the normal scope rules.

```
\savebox{cmd}(x_dimen, y_dimen) [pos]{text}
```

Same as the corresponding *\makebox* command, except the resulting box is saved in the bin *cmd*, which must be defined with *\newsavebox* (Section 6.4.3). Fragile.

```
\thinlines
\thicklines
```

They select one of the two standard thicknesses of lines and circles in the *picture* environment. The default is *\thinlines*. Robust.

```
\linethickness{len}
```

Declares the thickness of lines in a *picture* environment to be *len*, which must be a positive length. With the *pict2e* package, it applies to all lines; otherwise, it applies only to horizontal and vertical lines and does not affect the thickness of slanted lines and circles, or of the quarter circles drawn by *\oval* to form the corners of an oval.

### C.14.2 The *graphics* Package

The following commands are provided by the *graphics* package. They are all fragile. This package requires special support from the device driver.

`\scalebox{h_scale}[v_scale]{text}`

Produces a box by typesetting *text* in LR mode and scaling it horizontally by a factor of *h\_scale* and vertically by a factor of *v\_scale*. The default value of *v\_scale* is *h\_scale*.

`\resizebox {wdth}{ht}{text}`  
`\resizebox*{wdth}{ht}{text}`

Produce a box of width *wdth* and height *ht* by typesetting *text* in LR mode and scaling it horizontally and vertically to fit. In the \*-form, *ht* specifies the height + depth. If either argument is !, then the corresponding dimension is the one that maintains the aspect ratio of *text*.

`\rotatebox{ang}{text}`

Produces a box formed by typesetting *text* in LR mode and rotating it counterclockwise through an angle of *ang* degrees. The box is the smallest one containing the rotated box; its reference point is at the same height as that of the rotated box.

`\reflectbox{text}`

Produces a box by typesetting *text* in LR mode and reflecting it about a vertical line.

`\includegraphics [x_len1,y_len1][x_len2,y_len2]{file_name}`  
`\includegraphics* [x_len1,y_len1][x_len2,y_len2]{file_name}`

Produces a box containing the graphic material in the file named *file\_name*. With no optional arguments, the reference point and size of the box are specified by the file. The optional arguments specify a box of width *x\_len*<sub>2</sub> - *x\_len*<sub>1</sub> and height *y\_len*<sub>2</sub> - *y\_len*<sub>1</sub> whose reference point is shifted a distance of *x\_len*<sub>1</sub> to the right and *y\_len*<sub>1</sub> up from the lower-left corner of the contents of the file. Specifying only one optional argument is the same as giving a first optional argument of [0pt,0pt]. The \*-form clips the material by removing everything outside the specified box; the normal form does not.

### C.14.3 The color Package

The following commands are provided by the `color` package. They are all fragile. This package requires special support from the device driver.

`\definecolor{clr}{mdl}{val}`

Defines *clr*, which may be any sequence of letters and numbers, to be the name of the color specified by the color model *mdl* and color value *val*. L<sup>A</sup>T<sub>E</sub>X's standard color models are:

`gray` A color value is a number from 0 to 1 that specifies a shade of gray, where 0 is black.

`rgb` A color value is a list of three numbers from 0 to 1, separated by commas, that describe intensities of red, green, and blue light.

`cmyk` A color value is a list of four numbers from 0 to 1, separated by commas, that specify amounts of cyan, magenta, yellow, and black ink.

Other color models may also be supported. The package predefines the following color names: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, and `magenta`. All other color names must be defined before they are used.

`\color{clr}`

Declares `clr` to be the current text color. It obeys the normal scoping rules.

`\textcolor{clr}{text}`

Equivalent to `\color{clr} text`.

`\colorbox{bkgd_clr}{text}`

Produces a box by typesetting `text` in LR mode on a background of color `bkgd_clr`. The box includes a border of width `\fboxsep`.

`\fcolorbox{line_clr}{bkgd_clr}{text}`

Like `\colorbox`, except it also puts a line of width `\fboxrule` and color `line_clr` around the box.

`\pagecolor{clr}`

Declares `clr` to be the page's background color. It is a global declaration.

## C.15 Font Selection

A font is a size and style of type. A text font is selected by specifying the size and style. There are also special symbol fonts that are selected implicitly by math-mode commands. It is possible to select a font that is not available, in which case L<sup>A</sup>T<sub>E</sub>X types a warning and substitutes a similar font.

### C.15.1 Changing the Type Style

A type style is specified by three separate components: shape, series, and family. Changing one component does not affect the others. These components are changed by the following declarations, which obey the normal scope rules:

|                        |                        |                          |                      |
|------------------------|------------------------|--------------------------|----------------------|
| <code>\mdseries</code> | Medium Series          | <code>\upshape</code>    | Upright Shape        |
| <code>\bfseries</code> | <b>Boldface Series</b> | <code>\itshape</code>    | <i>Italic Shape</i>  |
| <code>\rmfamily</code> | Roman Family           | <code>\slshape</code>    | <i>Slanted Shape</i> |
| <code>\sffamily</code> | Sans Serif Family      | <code>\scshape</code>    | SMALL CAPS SHAPE     |
| <code>\ttfamily</code> | Typewriter Family      | <code>\normalfont</code> | Normal Style         |

The `\normalfont` declaration sets series, family, and shape to that of the document's main text font.

Each of these declarations has a corresponding command `\text...{text}` that typesets *text* in the scope of the declaration. The commands and their corresponding declarations are:

|                            |                        |                                |                          |
|----------------------------|------------------------|--------------------------------|--------------------------|
| <code>\textmd{text}</code> | <code>\mdseries</code> | <code>\textup{text}</code>     | <code>\upshape</code>    |
| <code>\textbf{text}</code> | <code>\bfseries</code> | <code>\textit{text}</code>     | <code>\itshape</code>    |
| <code>\textrm{text}</code> | <code>\rmfamily</code> | <code>\textsl{text}</code>     | <code>\slshape</code>    |
| <code>\textsf{text}</code> | <code>\sffamily</code> | <code>\textsc{text}</code>     | <code>\scshape</code>    |
| <code>\texttt{text}</code> | <code>\ttfamily</code> | <code>\textnormal{text}</code> | <code>\normalfont</code> |

Words typeset in typewriter style will not be hyphenated except where permitted by `\-` commands.

None of these commands or declarations can be used in math mode. They are all robust.

### C.15.2 Changing the Type Size

The following declarations select a type size, but leave the type style unaffected. They are listed in nondecreasing size; in some document-class options, two different size declarations may have the same effect.

|                            |                          |                     |                    |
|----------------------------|--------------------------|---------------------|--------------------|
| <code>\tiny</code>         | <code>\small</code>      | <code>\large</code> | <code>\huge</code> |
| <code>\scriptsize</code>   | <code>\normalsize</code> | <code>\Large</code> | <code>\Huge</code> |
| <code>\footnotesize</code> |                          | <code>\LARGE</code> |                    |

These commands may not be used in math mode; they are all fragile.

### C.15.3 Special Symbols

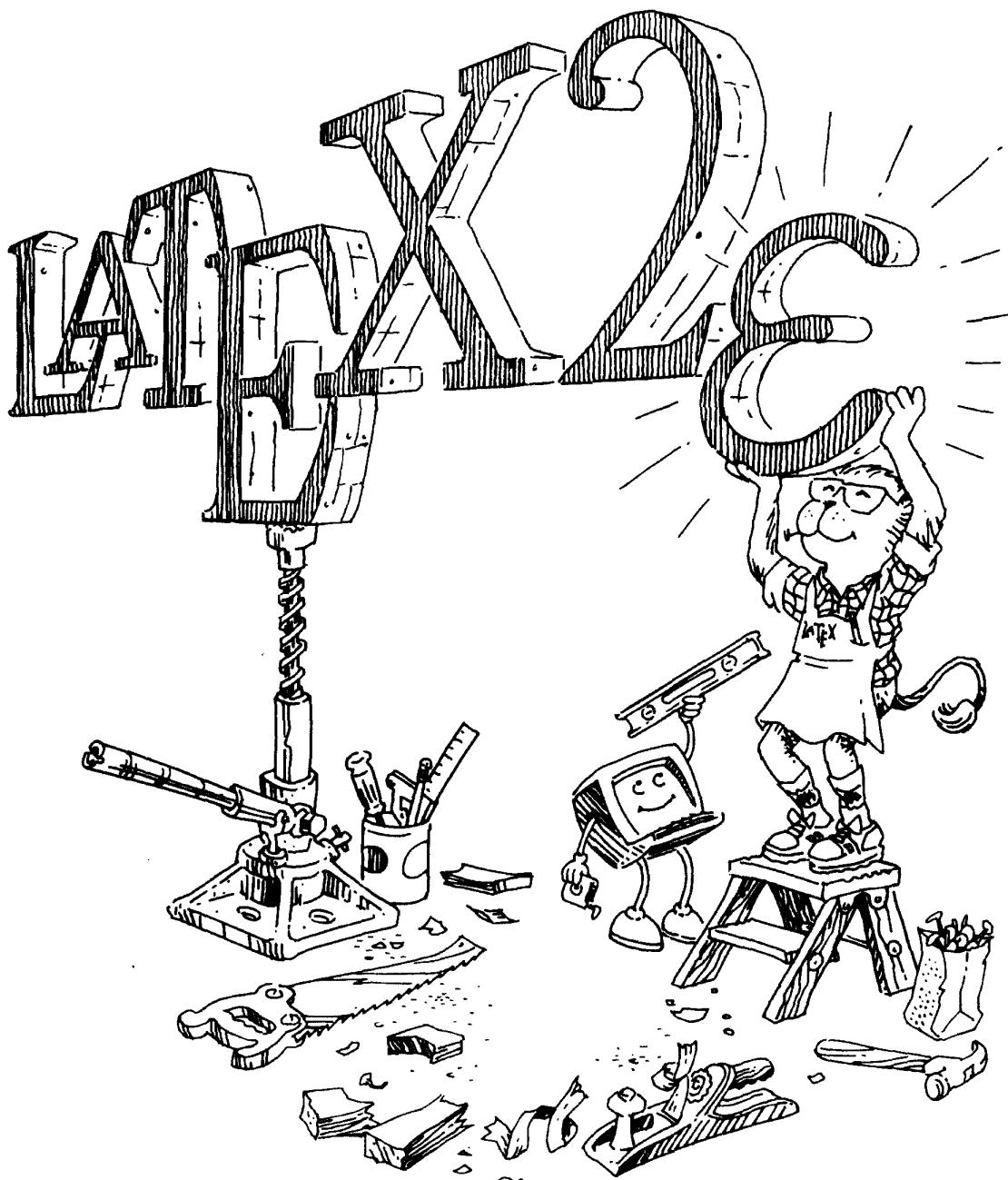
Special symbols can be obtained from special fonts. The *L<sup>A</sup>T<sub>E</sub>X Companion* explains how to get L<sup>A</sup>T<sub>E</sub>X to use such a font.

`\symbol{num}`

Chooses the symbol with number *num* from the current font. Octal (base 8) and hexadecimal (base 16) numbers are preceded by ' and ", respectively. Robust.

## APPENDIX D

# What's New



This appendix lists most of the differences between  $\text{\LaTeX}$  2.09, the original version of  $\text{\LaTeX}$ , and the current version,  $\text{\LaTeX}$  2 $\varepsilon$ .

## Document Styles and Style Options

Documents prepared for  $\text{\LaTeX}$  2 $\varepsilon$  begin with a `\documentclass` command (Section 2.2.2).  $\text{\LaTeX}$  2 $\varepsilon$  realizes it is processing a  $\text{\LaTeX}$  2.09 input file and enters *compatibility mode* when it encounters a  $\text{\LaTeX}$  2.09 `\documentstyle` command. Most  $\text{\LaTeX}$  2.09 input files will work with  $\text{\LaTeX}$  2 $\varepsilon$ . However, an error may occur if  $\text{\LaTeX}$  2 $\varepsilon$  reads an auxiliary file produced by  $\text{\LaTeX}$  2.09, so it's a good idea to delete such files before running  $\text{\LaTeX}$  2 $\varepsilon$ .

The document styles of  $\text{\LaTeX}$  2.09 have become document classes.  $\text{\SLiTeX}$  has been eliminated; slides are produced using the `slides` document class. Standard document-style options that controlled formatting, such as `twoside`, have become document-class options, and are specified as optional arguments to the `\documentclass` command. Other document-style options have become packages, loaded with the `\usepackage` command (Section 2.2.2). Most nonstandard document-style options will work as  $\text{\LaTeX}$  2 $\varepsilon$  packages.

## Type Styles and Sizes

The  $\text{\LaTeX}$  2.09 commands for changing type style, such as `\tt`, will still work more or less the same as before. The major difference is that `\sc` and `\sl` can no longer be used in math mode (except when  $\text{\LaTeX}$  2 $\varepsilon$  is in compatibility mode). However, instead of using these commands, you should switch to the more rational commands described in Sections 3.1 and 3.3.8 for changing type style. The new commands treat the different aspects of type style separately—for example, allowing you easily to specify bold sans serif type. The commands for changing type size are the same, but they no longer change the type style (except in compatibility mode). The `\boldmath` command now works better (Section 3.3.8).

## Pictures and Color

The `picture` environment has been enhanced by the addition of the `\qbezier` command for drawing curves (Section 7.1.3). The `pict2e` package also removes many restrictions on `picture` environment commands, such as limitations on the slopes of lines and arrows. The `graphics` package allows you to insert pictures produced by other programs (Section 7.2). It also defines commands for scaling and rotating text and pictures. The `color` package defines commands for producing colored text (Section 7.3).

## Other New Features

You can now define a command or environment that has an optional argument. The `\ensuremath` command is useful for defining a command that can appear in or out of math mode. See Section 3.4.

Control of the placement of floats (figures and tables) has been enhanced with a new float-location option `!` that encourages L<sup>A</sup>T<sub>E</sub>X to print the float as soon as possible, and with the `\suppressfloats` command to prevent additional floats on the current page.

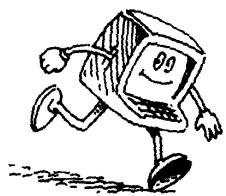
When sending your document electronically, you can bundle other files along with your input file using the `filecontents` environment (Section 4.7).

Analogs of the `\settowidth` command have been added for determining the dimensions of text other than the width (Section 6.4.1). In the length arguments of various box-making commands, it is now possible to refer to the dimensions of the text argument using commands such as `\width` (Section C.13.3).

The `ifthen` package defines commands for writing simple programs (Section C.8.5).

The `\enlargethispage` command has been added to help in correcting bad page breaks (Section 6.2.2). The `\samepage` command still works, but is now of little use.

A few commands and options have been added to the `book` document class (Section 5.1).



## APPENDIX E

# Using Plain T<sub>E</sub>X Commands



L<sup>A</sup>T<sub>E</sub>X is implemented as a TeX “macro package”—a series of predefined TeX commands. Plain TeX is the standard version of TeX, consisting of “raw” TeX plus the *plain* macro package. You can use Plain TeX commands to do some things that you can’t with standard L<sup>A</sup>T<sub>E</sub>X commands. However, before using Plain TeX, check the *L<sup>A</sup>T<sub>E</sub>X Companion* to see if there is a package that does what you want.

Most Plain TeX commands can be used in L<sup>A</sup>T<sub>E</sub>X, but only with care. L<sup>A</sup>T<sub>E</sub>X is designed so its commands fit together as a single system. Many compromises have been made to ensure that a command will work properly when used in any reasonable way with other L<sup>A</sup>T<sub>E</sub>X commands. A L<sup>A</sup>T<sub>E</sub>X command may not work properly when used with Plain TeX commands not described in this book.

There is no easy way to tell whether a Plain TeX command will cause trouble, except by trying it. A general rule is not to combine a L<sup>A</sup>T<sub>E</sub>X command or environment with Plain TeX commands that might modify parameters it uses. For example, don’t use a Plain TeX command such as `\hangindent` that modifies TeX’s paragraph-making parameters inside one of L<sup>A</sup>T<sub>E</sub>X’s list-making environments.

You should not modify any parameters that are used by L<sup>A</sup>T<sub>E</sub>X’s `\output` routine, except as specified in this book. In particular, you should forget about most of Chapter 15 of *The TeXbook*. However, L<sup>A</sup>T<sub>E</sub>X does obey all of TeX’s conventions for the allocation of registers, so you can define your own counts, boxes, etc., with ordinary TeX commands.

Listed below are all the Plain TeX commands whose definitions have been eliminated or changed in L<sup>A</sup>T<sub>E</sub>X. Not listed are L<sup>A</sup>T<sub>E</sub>X commands that approximate the corresponding Plain TeX versions, and some “internal” commands whose names contain @ characters.

### Tabbing Commands

The following commands are made obsolete by L<sup>A</sup>T<sub>E</sub>X’s tabbing environment:

|                      |                         |                        |                 |
|----------------------|-------------------------|------------------------|-----------------|
| <code>\tabs</code>   | <code>\tabsdone</code>  | <code>\settabs</code>  | <code>\+</code> |
| <code>\tabset</code> | <code>\cleartabs</code> | <code>\tabalign</code> |                 |

### Output, Footnotes, and Figures

The following commands that require Plain TeX’s output routine are obsolete. They have been replaced by L<sup>A</sup>T<sub>E</sub>X’s footnote-making commands and its `figure` and `table` environments.

|                            |                             |                            |                          |
|----------------------------|-----------------------------|----------------------------|--------------------------|
| <code>\pageno</code>       | <code>\nopagenumbers</code> | <code>\makeheadline</code> | <code>\topins</code>     |
| <code>\headline</code>     | <code>\advancepageno</code> | <code>\makefootline</code> | <code>\topinsert</code>  |
| <code>\footline</code>     | <code>\pagebody</code>      | <code>\dosupereject</code> | <code>\midinsert</code>  |
| <code>\normalbottom</code> | <code>\plainoutput</code>   | <code>\footstrut</code>    | <code>\pageinsert</code> |
| <code>\folio</code>        | <code>\pagecontents</code>  | <code>\vfootnote</code>    | <code>\endinsert</code>  |

## Font-Selecting Commands

The following Plain *T<sub>E</sub>X* commands are not defined in *L<sup>A</sup>T<sub>E</sub>X*:

|         |          |           |
|---------|----------|-----------|
| \fivei  | \fivebf  | \sevensy  |
| \fiverm | \seveni  | \teni     |
| \fivesy | \sevenbf | \oldstyle |

## Aligned Equations

The following Plain *T<sub>E</sub>X* commands have been made obsolete by the *eqnarray* and *eqnarray\** environments:

|          |            |             |
|----------|------------|-------------|
| \eqalign | \eqalignno | \leqalignno |
|----------|------------|-------------|

## Miscellaneous

Plain *T<sub>E</sub>X*'s *\$\$* does not work properly; it has been replaced by the *L<sup>A</sup>T<sub>E</sub>X* commands *\[* and *\]*. Plain *T<sub>E</sub>X*'s *\begin{section}* command has been replaced by *L<sup>A</sup>T<sub>E</sub>X*'s sectioning commands; its *\end* and *\bye* commands have been replaced by *\end{document}*. The Plain *T<sub>E</sub>X* commands *\centering* and *\line* have had their names usurped by *L<sup>A</sup>T<sub>E</sub>X* commands, and the syntax of the *\input* command has been changed to conform to *L<sup>A</sup>T<sub>E</sub>X* conventions. Most functions performed by Plain *T<sub>E</sub>X*'s *\line* command can be achieved by the *center*, *flushleft*, and *flushright* environments. The *\magnification* command of Plain *T<sub>E</sub>X* has no counterpart in *L<sup>A</sup>T<sub>E</sub>X*. Magnification of the output can often be done by the program that prints the *dvi* file.



# Bibliography

- [1] Theodore M. Bernstein. *The Careful Writer: A Modern Guide to English Usage*. Atheneum, New York, 1965.
- [2] *The Chicago Manual of Style*. University of Chicago Press, fourteenth edition, 1993.
- [3] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L<sup>A</sup>T<sub>E</sub>X Companion*. Addison-Wesley, Reading, Massachusetts, 1994.
- [4] Donald E. Knuth. *The T<sub>E</sub>Xbook*. Addison-Wesley, Reading, Massachusetts, 1994.
- [5] N. E. Steenrod, P. R. Halmos, M. M. Schiffer, and J. A. Dieudonné. *How to Write Mathematics*. American Mathematical Society, London, 1983.
- [6] William Strunk, Jr. and E. B. White. *The Elements of Style*. Macmillan, New York, third edition, 1979.
- [7] Mary-Claire van Leunen. *A Handbook for Scholars*. Oxford University Press, New York, Oxford, revised edition, 1992.
- [8] *Words Into Type*. Prentice-Hall, Englewood Cliffs, New Jersey, third edition, 1974.

This page intentionally left blank

# Index

□ (space character), 13, 32  
  ignored in math mode, 36, 50  
  in LR mode, 36  
  printing, 64  
\\□ (interword space), 14, 16, 32, 33, 51, 170  
  used with `\thanks`, 181  
# (hash mark), 12, 32, 166  
  in definition, 54, 56, 192, 193  
  misplaced, 141, 145  
  printing, 15, 171  
\\# (#), 15, 39, 171  
\$ (dollar sign), 12, 32, 166  
  delimiting formula, 18, 33, 39, 187  
  delimits scope of declaration, 168  
  missing, 142  
  not fragile, 22  
  printing, 15, 171  
  unmatched, 141  
\\\$ (\$), 15, 39, 171  
\$\$ (TeX command), 233  
% (percent sign), 12, 19, 32, 166  
  for ending line without adding space, 33, 109  
  in `\index` argument, 154  
  printing, 15, 171  
\\% (%), 15, 39, 171  
& (ampersand), 12, 32, 166  
  in `array` or `tabular` environment, 45, 206  
  misplaced, 141  
  printing, 15, 171  
  too many in row, 141  
\\& (&), 15, 39, 171  
~ (tilde), 12, 17, 32, 33, 166, 170  
  used with `\ref` and `\pageref`, 68  
\\~ (˜ accent), 38

\_ (underscore), 12, 32, 166  
  for subscript, 18, 33, 40, 189  
  printing, 15, 171  
\\\_ (-), 15, 39, 171  
^ (circumflex), 12, 32, 166  
  for superscript, 18, 33, 40, 189  
\\^ (˜ accent), 38  
\\ (backslash), 12, 16, 32, 166  
\\\\ (new line), 25–26, 34, 169–170, 213  
  [ after, 26  
  \* after, 167  
  after last row of `array` or `tabular`, 62, 206  
  bad use of, 147  
  between paragraphs, 139  
  in `\address` argument, 84  
  in `array` environment, 45, 169  
  in `\author` argument, 169  
  in `center` environment, 111, 169  
  in `eqnarray` environment, 47, 169  
  in `flushleft` environment, 169  
  in `flushright` environment, 169  
  in `p` column of `array` or `tabular`, 205  
  in paragraph mode, 96, 213  
  in `\shortstack` argument, 169  
  in `\signature` argument, 84  
  in `tabbing` environment, 61, 169, 202  
  in `tabular` environment, 62, 169  
  in `\title` argument, 181  
  in `verse` environment, 169, 184  
  is fragile, 33  
  missing, 141  
  optional argument of, 167  
  two in a row, 169  
\\\\\* (new line), *see* \\\\  
{ (left brace), 12, 32  
  enclosing argument, 33, 166

Index Order

□ (space)  
#  
\$  
%  
&  
~ (tilde)  
\_ (underscore)  
^ (circumflex)  
\\ (backslash)  
{  
}  
. (period)  
: (colon)  
; (semicolon)  
, (comma)  
?  
!  
‘ (left quote)  
’ (right quote)  
(  
)  
[  
]  
- (dash)  
/ (slash)  
\*  
@  
+  
=  
| (vertical line)  
<  
>  
" (double quote)  
0 ... 1  
A a B ... z

- Index Order**
- in **bib** file, 158
  - in **\index** argument, 75
  - missing, 142
  - printing, 15, 171
  - scope delimited by, 27–28
  - \{ ( {)**, 15, 39, 47, 171
    - in **bib** file, 156
    - in **\index** argument, 212
  - \} ( right brace)**, 12, 32
    - enclosing argument, 33, 166
    - in **bib** file, 158
    - in **\index** argument, 75
    - missing, 142
    - printing, 15, 171
    - scope delimited by, 27–28, 168
    - unmatched, 141
  - \} ( {)**, 15, 39, 47, 171
    - in **bib** file, 156
    - in **\index** argument, 212
  - \. ( period)**, 12, 32, 33
    - invisible delimiter, 47
    - ..., see ellipsis
    - \. (` accent)**, 38
    - \. (colon)**, 12, 32
    - \: (medium space)**, 51, 191
    - \; (semicolon)**, 12, 32
    - \; (thick space)**, 51, 191
    - \, (comma)**, 12, 32
      - not allowed in citation key, 69
    - \, (thin space)**, 14, 33, 51, 170, 191
    - \? (question mark)**, 12, 32, 33
      - in error message, 29
    - \?' ( ?)**, 39
    - \From**, 32
      - \! (exclamation point)**, 12, 32, 33
        - in error message, 29
      - in **figure** or **table** argument, 197
      - in **\index** argument, 152
      - in **\resizebox** argument, 129, 224
      - \! ( ?)**, 39
      - \! (negative thin space)**, 51, 191
      - \!h** float specifier changed, 146
      - \` (left quote)**, 12, 13, 32, 33, 170
      - \` (` accent or tabbing command)**, 38, 203
        - in **parbox**, 203
    - \` (right quote)**, 12, 13, 32, 33, 170
      - in formula, 18, 189
      - period before, 15
      - specifying octal number, 226
    - \` (` accent or tabbing command)**, 38, 203
      - in **parbox**, 203
    - \( ( left parenthesis)**, 12, 32
      - delimiter, 47
      - in terminal output, 134
    - \( ( begin formula)**, 18, 33, 39, 187
      - in **ifthen** package expression, 196
      - in math mode error, 136
      - is fragile, 22, 33
    - \) ( right parenthesis)**, 12, 32
      - delimiter, 47
      - in terminal output, 134
      - period before, 15
    - \) ( end formula)**, 18, 33, 187
      - delimits scope of declaration, 168
      - in **ifthen** package expression, 196
      - is fragile, 22, 33
      - not in math mode error, 136
    - \[ ( left bracket)**, 12, 32, 33, 166
      - ambiguous, 25
      - delimiter, 47
      - printed on terminal, 135
    - \[ ( begin displayed formula)**, 26, 34, 39, 188
      - in math mode error, 136
      - is fragile, 33
    - \] ( right bracket)**, 12, 32, 33, 166
      - ambiguous, 25
      - delimiter, 47
      - delimiting optional argument, 168
      - in **\item** argument, 167
      - printed on terminal, 135
    - \] ( end displayed formula)**, 26, 34, 39, 188
      - delimits scope of declaration, 168
      - is fragile, 33
      - not in math mode error, 136
    - \- (dash or minus)**, 12, 14, 32, 33, 170
      - in overfull **\hbox** message, 93
      - space around, 48
      - unary, 48
      - (number-range dash), 14, 33, 170

|                                                   |                                                     |                    |
|---------------------------------------------------|-----------------------------------------------------|--------------------|
| --- (punctuation dash), 14, 33, 170               | \= (‐ accent or tabbing command), 38, 61, 201, 202  | <b>Index Order</b> |
| \- (hyphenation or tabbing command), 93, 202, 213 | in <b>parbox</b> , 203                              |                    |
| error in, 140                                     | too many, 139                                       |                    |
| in <b>parbox</b> , 203                            | (vertical line), 12, 32                             |                    |
| in <b>tabbing</b> environment, 203                | delimiter, 47                                       |                    |
| instead of <b>\hyphenation</b> , 143              | in <b>array</b> or <b>tabular</b> argument, 62, 205 |                    |
| needed in typewriter style, 226                   | in <b>array</b> or <b>tabular</b> argument, 207     |                    |
| / (slash), 12, 32                                 | < (less than), 12, 32                               |                    |
| delimiter, 47                                     | in <b>ifthen</b> package expression, 195            |                    |
| quotient symbol, 40, 51                           | in <b>\lengthtest</b> expression, 196               |                    |
| * (asterisk), 12, 32, 167                         | \< (tabbing command), 202                           |                    |
| acts like optional argument, 26                   | error in, 140                                       |                    |
| after command name, 26                            | > (greater than), 12, 32                            |                    |
| argument to <b>\nocite</b> , 70                   | in <b>ifthen</b> package expression, 195            |                    |
| in <b>array</b> or <b>tabular</b> argument, 205   | in <b>\lengthtest</b> expression, 196               |                    |
| written on terminal, 30                           | \> (tabbing command), 61, 201, 202                  |                    |
| *-expression, 205                                 | error in, 140                                       |                    |
| *-form                                            | " (double quote), 12                                |                    |
| of command, 26, 33, 167                           | in <b>bib</b> file, 156                             |                    |
| of environment, 167                               | in index entry, 154                                 |                    |
| of sectioning command, 174                        | specifying hexadecimal number, 226                  |                    |
| @ (at sign), 12, 32                               | \" (‐ accent), 38                                   |                    |
| command name with, 91, 166                        | 0 (zero), 12                                        |                    |
| in <b>\index</b> argument, 153                    | incorrect use as length, 99, 141                    |                    |
| regarded as letter in <b>sty</b> file, 91         | 1 (one), 12                                         |                    |
| \@, 14, 33, 170                                   | 10pt document-class option, 177                     |                    |
| @-expression, 205                                 | 11pt document-class option, 19, 115, 177            |                    |
| error in, 145                                     | 12pt document-class option, 19, 115, 177            |                    |
| fragile command in, 168                           | 2.09, 2                                             |                    |
| missing, 138                                      | 2 $\varepsilon$ , 2                                 |                    |
| \@array, 145                                      | \a' (‐ accent in <b>tabbing</b> environment), 203   |                    |
| \@chapter, 92                                     | \a' (‐ accent in <b>tabbing</b> environment), 203   |                    |
| \@makechapterhead, 92                             | \a= (‐ accent in <b>tabbing</b> environment), 203   |                    |
| \@schapter, 92                                    | A4 paper size, 177                                  |                    |
| @string, 159                                      | a4paper document-class option, 177                  |                    |
| + (plus), 12, 32                                  | A5 paper size, 177                                  |                    |
| space around, 48                                  | a5paper document-class option, 177                  |                    |
| unary, 48                                         | \AA (‐), 39                                         |                    |
| \+ (tabbing command), 202                         | \aa (‐), 39                                         |                    |
| error in, 140                                     | abbreviation, 14                                    |                    |
| in Plain <b>T<sub>E</sub>X</b> , 232              | in <b>bib</b> file, 158–159                         |                    |
| = (equals), 12, 32                                |                                                     |                    |
| in <b>bib</b> file, 156                           |                                                     |                    |
| in <b>ifthen</b> package expression, 195          |                                                     |                    |
| in <b>\lengthtest</b> expression, 196             |                                                     |                    |

- abrv** bibliography style, 70  
**\abovedisplayshortskip**, 189  
**\abovedisplayskip**, 188  
**abstract**, 90, 181–183  
     not in **book** document class, 183  
     on separate page, 88, 177  
**abstract** environment, 90, 183  
     effect of **titlepage** option, 88  
**accent**, 38  
     dotless i and j for, 38, 50, 190  
     in **bib** file, 158  
     in **tabbing** environment, 203  
     math mode, 49–50, 190  
     not available in typewriter style, 38  
     wide math, 49  
**acknowledgment** of support, 181  
**\acute** (‘math accent), 50  
**\addcontentsline**, 175  
     argument too long, 143  
     moving argument of, 168  
**address** field (in **bib** file), 162  
**\address**, 84  
**\addtime**, 83  
**\addtocontents**, 67, 176  
     argument too long, 143  
     moving argument of, 168  
**\addtocounter**, 98, 194  
     error in, 138  
     **\protect** not used in argument of, 168  
     scope of, 98, 168  
**\addtolength**, 101, 216  
**\addvspace**, 217  
**\advancepageno** (TeX command), 232  
**\AE** (Æ), 39  
**\ae** (æ), 39  
**\aleph** (ℵ), 43  
**aligning**  
     formulas on left, 88, 177  
     in columns, 60–63, 201–207  
**alignment tab** error, 141  
**alltt** environment, 187  
**alltt** package, 178, 187  
**\Alpha**, 98, 195  
**\alpha**, 98, 195  
**alpha** bibliography style, 70  
**\alpha** (α), 41  
**alphabetic** page numbers, 180  
**\amalg** (II), 42  
**ambiguous** [ or ], 25  
**American Mathematical Society**, 52  
**ampersand**, *see &*  
**amstex** package, 52, 178  
**and** separating names in **bib** file field,  
     158  
**\and**, 21, 34, 181  
     in **ifthen** package expression, 196  
**and others**, 158  
**\angle** (∠), 43  
**annote** field (in **bib** file), 162  
**apostrophe**, 14, 170  
**appendix**, 22, 175  
**\appendix**, 22, 175  
**\approx** (≈), 43  
**arabic** page numbers, 89, 180  
**\arabic**, 98, 195  
**\arccos** (arccos), 44  
**\arcsin** (arcsin), 44  
**\arctan** (arctan), 44  
**\arg** (arg), 44  
**argument** (of command), 16, 33, 166–167  
     braces enclosing, 33, 55, 166  
     coordinate pair as, 119  
     mandatory, 166  
     missing, 142  
     *in thebibliography* environment,  
       138  
     moving, *see moving argument*  
     of picture command, 119  
     omitted, 141  
     optional, *see optional argument*  
     positioning, 205, 217–218, 221–222  
     processed multiple times, 110  
     scope of declaration in, 27, 168  
**\verb** in, 140  
**array**, 45–46  
**array** environment, 45–46, 204–207  
     \\ in, 45, 169  
     box made by, 103  
     error in, 137, 138, 141, 145  
     extra space around, 205  
     illegal character in argument, 137  
     interrow space in, 169  
     large, 143  
     making symbol with, 42

- scope of declaration in, 45  
strut in, 169  
versus `tabular`, 46, 60, 62  
`\arraycolsep`, 207  
`\arrayrulewidth`, 207  
`\arraystretch`, 207  
arrow  
  accent, *see* `\vec`  
  in formula, 53  
  in margin, 59  
  in picture, 123, 222  
  symbols, 43  
  zero-length, 123  
arrowhead, 123  
`article` bibliography entry type, 161  
`article` document class, 19, 176  
  appendix in, 175  
  `\chapter` not defined in, 21, 174  
  default page style, 89  
  `thebibliography` environment in, 209  
  used in examples, 21  
`ASCII` file, 144  
assumption, 56  
`\ast` (\*), 42  
`\asymp` ( $\asymp$ ), 43  
at sign, *see* `\@`  
author, 20  
`author` field (in `bib` file), 162  
author's address in title, 181  
`\author`, 20, 34, 181  
  `\backslash` in argument, 169  
  missing, 146  
authors, multiple, 21  
`aux` file, 208  
  entry generated by `\label`, 209  
  entry written by `\cite` and `\nocite`, 210  
  error when reading, 135, 138  
`auxiliary` file, error reading, 135, 228  
`axiom`, 56  
  
`\b` (bottom)  
  float specifier, 197  
  oval-part argument, 124, 223  
  positioning argument, 46, 105, 121, 218, 222  
`\b` ( \_ accent), 38  
  
B5 paper size, 177  
`b5paper` document-class option, 177  
`babel` package, 38, 94, 178  
back matter (of a book), 80  
`\backmatter`, 80  
backslash, 12, 16, 32, 166  
`\backslash` (), 43, 47  
backspace, 51, 101  
bar over a symbol, 49  
`\bar` (  $\bar{}$  math accent), 50  
`\baselineskip`, 100, 172  
`\baselinestretch`, 172  
`\batchmode`, 30  
`bbi` file, 71, 156, 208  
  read by `\bibliography`, 209  
`\begin`, 23, 167  
  delimits scope of declaration, 27–28  
  is fragile, 22, 33  
  of nonexistent environment, 137  
  unmatched, 136  
`\begin{document}`, 19, 170  
  aux file read by, 208  
  error while processing, 135  
  extra, 136  
  missing, 138  
`\begin{section}` (TeX command), 233  
`\belowdisplayshortskip`, 189  
`\belowdisplayskip`, 189  
`\beta` ( $\beta$ ), 41  
Bezier curve, 125–126, 221  
`\bfseries`, 37, 226  
`bib` file, 70, 155–164  
  abbreviations in, 158–159  
  entry, 156  
  keeping data in, 161  
  specified by `\bibliography`, 156, 209  
`\bibindent`, 178  
`\bibitem`, 71, 210  
  moving argument of, 72, 168  
`bibliographic` database, *see* `bib` file  
`bibliography`, 69–72, 209–210  
  open format, 71, 177  
  produced by `thebibliography`  
  environment, 209  
  style, 70–71  
`\bibliography`, 70, 209  
  `bbi` file read by, 208

- specifies `bib` files, 156  
`\bibliographystyle`, 70  
`BIBTeX`, 69–71, 155–164  
`bbl` file written by, 208  
 producing bibliography with, 209  
 big delimiter, 46  
`\bigcap` ( $\bigcap$ ), 44  
`\bigcirc` ( $\bigcirc$ ), 42  
`\bigcup` ( $\bigcup$ ), 44  
`\bigodot` ( $\bigodot$ ), 44  
`\bigoplus` ( $\bigoplus$ ), 44  
`\bigotimes` ( $\bigotimes$ ), 44  
`\bigskip`, 216  
`\bigskipamount`, 216  
`\bigsqcup` ( $\bigcup$ ), 44  
`\bigtriangledown` ( $\bigtriangledown$ ), 42  
`\bigtriangleup` ( $\bigtriangleup$ ), 42  
`\biguplus` ( $\biguplus$ ), 44  
`\bigvee` ( $\bigvee$ ), 44  
`\bigwedge` ( $\bigwedge$ ), 44  
 black, 132, 225  
 blank line, 13, 33  
     above or below environment, 23  
     before displayed formula, 26  
     in formula, 142, 171  
     in input, 166  
     in sectioning command, 171  
`\par` equivalent to, 171  
     paragraph-ending, 95, 166, 171  
 blank page, 97  
     made by `\cleardoublepage`, 215  
     with `titlepage` environment, 180  
 blob of ink, rectangular, 106  
 blue, 132, 225  
`\bmod`, 44, 190  
 body, page, 89, 179  
     height of, 100  
 boldface type series, 37, 226  
     in math mode, 51, 191  
`\boldmath`, 51, 191  
     font warning caused by, 52, 145  
 book, 80  
`book` bibliography entry type, 161  
`book` document class, 80, 176  
     appendix in, 175  
     no abstract in, 183  
     `thebibliography` environment in, 209  
`\booklet` bibliography entry type, 161  
`booktitle` field (in `bib` file), 162  
 boolean register, 196  
`\boolean`, 196  
`\bot` ( $\bot$ ), 43  
 bottom of line, 107  
`\bottomfraction`, 200  
`\bottomnumber` counter, 199  
`\bowtie` ( $\bowtie$ ), 43  
 box, 103–108, 217  
     dashed, 121, 222  
     declaration local to, 103  
     dimensions of, 103  
     displaying, 104  
     framed, 104, 217  
         in `picture` environment, 222  
     in formula, 103  
     LR, 103–104, 107  
     rule, 103, 106  
     saving, 107, 127, 218  
     typeset in paragraph mode, 104  
     with specified width, 104  
     zero-width, 121  
`\Box` ( $\Box$ ), 43  
 boxes, 217–219  
     formatting with, 108–111  
 brace, curly, 12  
     enclosing command argument, 33, 55, 166  
     error caused by unbalanced, 136  
     horizontal, 49  
     in `bib` file, 158  
     in `\index` argument, 75, 153  
     missing, 142  
     removed from first argument of  
         `\newcommand`, 92  
 bracket, square, 12  
     enclosing optional argument, 20, 166  
     mistaken for optional argument, 142  
     printed on terminal, 135  
 break, line, 93–96, 213–214  
     interword space without, 17, 170  
     permitting, 95  
     preventing, 17, 95, 171  
     with `\`, 26  
 break, page, 96–97, 214–215  
     bad, 147

- in **tabbing** environment, 60  
**\breve** (ˇ math accent), 50  
Brinch Hansen, Per, 157  
buffer size, 143  
bug, L<sup>A</sup>T<sub>E</sub>X, 139  
**\bullet** (•), 42  
Bush, George, 24  
**\bye** (TeX command), 233
- c (center) positioning argument, 45, 205  
**\c** (, accent), 38  
calligraphic letter, 42  
calligraphic type style, 51, 191  
**\cap** (∩), 42  
capacity exceeded, 142–144  
caps, small, 37, 226  
caption  
  cross-reference to, 68  
  figure or table, 58  
  multiple, 59  
**\caption**, 58, 198  
  argument too long, 143  
  fragile command in, 135  
  in **parbox**, 199  
  **\label** in argument, 209  
  list of figures or tables entry, 175, 208  
  moving argument of, 58, 168  
  precedes **\label**, 68  
case of letters  
  in command name, 16, 32  
  in key, 68, 69  
catching errors with text editor, 32  
**\cc**, 85  
**\cdot** (·), 42  
**\cdots** (· · ·), 40, 189  
**\center** environment, 111  
  \\ in, 111, 169  
  as displayed paragraph, 183  
  as list, 112  
  displaying a box with, 104  
  in title page, 90  
  **tabular** environment in, 63  
center line of formula, 46  
centered  
  array column, 45  
  ellipsis, 40, 189  
centering a figure or table, 112  
**\centering**, 112  
  in p column, 205  
  TeX command, 233  
centimeter (cm), 99, 215  
chapter, in separate file, 74  
**chapter** counter, 97  
**chapter** field (in **bib** file), 163  
**\chapter**, 21, 174  
  changes current page style, 89  
  in front and back matter of a book, 80  
  not in **article** document class, 174  
  uses **\clearpage**, 97  
character  
  code, 116  
  control, 144  
  end of line, 12  
  illegal, in **array** or **tabular**, 137  
  input, 32  
  invalid, 144  
  invisible, 12, 32  
  nonprinting, 144  
  punctuation, 12, 32  
    in key, 68  
  space, *see* space character  
  special, *see* special character  
**\check** (ˇ math accent), 50  
**chgsam.tex** file, 9  
**\chi** (χ), 41  
**\circ** (o), 42  
circle, 124, 222  
**\circle**, 124, 222  
**\circle\***, 124, 222  
circular reference, 263  
citation, 69–72, 209–210  
  key, 69, 71, 156  
  undefined, 145, 147  
**\cite**, 69, 210  
  wrong number printed by, 146  
class, document, 19–20  
  nonexistent, 137  
**classes.dtx** file, 92  
**\cleardoublepage**, 97, 215  
  figures and tables output by, 198  
**\clearpage**, 97, 215  
  checking capacity exceeded error, 143  
  figures and tables output by, 198  
  used by **\chapter**, 97

- used by `\include`, 97
- used by `\onecolumn`, 181
- used by `\twocolumn`, 180
- `\cleartabs` (TeX command), 232
- `\cline`, 62, 206
- clipping, 224
- `clock` document-class option, 83
- `\closing`, 85
- `cls` (class) file, missing, 137
- `\clubsuit` (♣), 43
- `cm` (centimeter), 99, 215
- `cmyk` color model, 132, 225
- code, character, 116
- colon, *see* :
- color, 131–132
  - background, 132
  - model, 132, 224
    - undefined, 140
  - undefined, 140
  - value, 224
- `color` package, 131–132, 178, 224–225
- `\color`, 131, 225
- `\colorbox`, 132, 225
- column
  - aligning text in, 60–63, 201–207
  - of text in picture, 123
- `\columnsep`, 178
- `\columnseprule`, 178
- comma (,), 12, 32
  - not allowed in citation key, 69
- command, 32–33, 166–170
  - \*-form of, 26, 33, 167
  - adding to table of contents, etc., 176
- argument, *see* argument
- definition, 7, 53–55, 192
  - # in, 54
  - in preamble, 55
  - space character in, 54
  - with optional argument, 192
  - with `\typein`, 212
- environment made from, 108
- form, 166
- fragile, 22, 25, 26, 33, 167–168
  - in @-expression, 168
- invisible, 169
- length, 99
- defining, 101
- not preceded by `\protect`, 168
- name, 16, 32, 166–167
  - \* after, 26
  - already used error, 137
  - beginning with `\end`, 55
  - case of letters in, 16, 32
  - correcting misspelled, 144
  - ending with space or end of line, 16
  - misspelled, 31
  - one-character, 166
  - with @, 91, 166
- names, too many, 143, 144
- nested too deeply, 144
- nesting, 16
- parameter, 54, 192
- picture, argument of, 119
- Plain TeX, 231–233
- preloaded, 91
- printing on terminal, 212
- redefining, 55
- robust, 22, 167
- sectioning, *see* sectioning command
- spacing, 170
- text-generating, 15
- with two optional arguments, 166
- commas, ellipsis between, 40
- comment, 19, 166
  - printed on terminal, 76
- common errors, 31
- compatibility mode, 228
- compressed bibliography style, 71
- Computer Modern font, 115
- computer program, formatting, 60
- concept index, 150
- concepts versus notation, 31
- `\cong` (≡), 43
- conjecture, 56
- contents, table of, 66–67, 175–176
  - adding commands to, 67
- control character, 144
- control sequence error, 142, 144
- coordinate, 118, 219
  - geometry, 118
  - grid, 126–127
  - local, 129
  - pair, 118
    - as argument, 119

- \coprod (Π), 44  
copy editing, double-spacing for, 172  
\copyright (©), 39, 173  
corner, rounded, 124–125  
\cos (cos), 44  
\cosh (cosh), 44  
\cot (cot), 44  
\coth (coth), 44  
counter, 97–99  
    command, error in, 144  
    created by \newtheorem, 97  
    creating a new, 99  
    for theorem-like environment, 97, 193  
    list, 114  
    reset by \stepcounter and  
        \refstepcounter, 194  
    too large, 137  
    value of, 98, 194  
cross-reference, 66–69, 209  
    in bib file, 159–160  
    in index, 152  
    information, printing, 208  
    labels, too many, 143, 144  
    to page number, 68  
    use of aux file for, 208  
crossref field (in bib file), 163  
\csc (csc), 44  
\cup (U), 42  
curly brace, *see* brace, curly  
curve, Bezier, 125–126, 221  
customizing style, 91–93  
cyan, 132, 225
- \d ( accent), 38  
\dag (†), 39, 173  
\dagger (†), 42  
dash, 14, 33, 170  
    intraword, 14, 170  
    number-range, 14  
    punctuation, 14  
\dashbox, 121, 221  
dashed box, 121, 222  
\dashv (⊣), 43  
data, keeping in bib file, 161  
database, bibliographic, *see* bib file  
date, 20  
    generating with \today, 15, 171  
    in title, 20  
    in title page, 90  
    on letter, 86  
\date, 20, 34, 181  
    \\ in, 169  
\dblfloatpagefraction, 200  
\dblfloatsep, 200  
\dbltextfloatsep, 200  
\dbltopfraction, 200  
dbltopnumber counter, 200  
\ddag (‡), 39, 173  
\ddagger (‡), 42  
\ddot (math accent), 50  
\ddots (‥), 41, 189  
declaration, 27–28, 33, 168  
    color, 131, 225  
    global, 168  
    local to a box, 103  
    picture, 223  
    scope of, *see* scope  
    type-size changing, 115  
declarations, file of, 73  
\def (TeX command), 92, 93  
default  
    argument, 166  
    page style, 179  
    type size, 115  
\definecolor, 132, 224  
defined, multiply, 146  
definition  
    command, *see* command  
    environment, *see* environment  
    recursive, 54  
    use doesn't match, 145  
\deg (deg), 44  
delimiter, 46–47, 190  
    bad, 136  
    unmatched math mode, 141  
\Delta (Δ), 41  
\delta (δ), 41  
depth  
    changing apparent, 107  
    of a box, 103  
\depth, 219  
description environment, 24–25, 34,  
    184  
    as list, 112

- item label overprinting text in, 184  
 used for glossary, 76
- design  
 logical, 7  
 typographic, 91  
 visual, 7, 88
- designer, typographic, 5
- \det** (det), 44  
 subscript of, 190
- determinant, 45
- device driver, 118
- device-independent file, *see* **dvi** file
- diacritical mark, *see* accent
- diagonal ellipsis, 40, 189
- \Diamond** (◊), 43  
**\diamond** (◊), 42  
**\diamondsuit** (◊), 43
- dictionary, exception, 143
- differential, 50
- digit, 12, 32
- \dim** (dim), 44
- dimen (**TeX** term), 100
- dimensional unit, 215
- disk (filled circle), 124, 222
- diskette, sending document on, 77
- display math style, 52, 188, 191
- displayed  
 formula, 26, 34, 39  
 blank line before, 26  
 math style for, 52, 191  
 multiline, 47–49  
 numbered, 39  
 space above and below, 107, 183,  
 188, 189, 218
- paragraph, 183–187  
 quotation, 23, 34  
 text, 23–26
- displaying a box, 104
- displaymath** environment, 26, 34, 39,  
 188  
 anomalous vertical space after, 183  
 displaying a box with, 104  
 size of symbols in, 42
- \displaystyle**, 52, 191
- distance, *see* length
- \div** (÷), 42
- document  
 class, 19–20, 176–178  
 nonexistent, 137  
 multilanguage, 38  
 non-English, 38  
 structure of, 170  
 style (**L<sup>A</sup>T<sub>E</sub>X 2.09**), 228
- document** environment, 34
- document-class option, 19, 88, 176  
 unused, 147
- document-style option (**L<sup>A</sup>T<sub>E</sub>X 2.09**), 228
- \documentclass**, 19, 34, 170, 176–178  
 in **\input** file, 73  
 missing, 144  
 unused option, 147
- \documentstyle** (**L<sup>A</sup>T<sub>E</sub>X 2.09** command),  
 228
- \dosupereject** (**TeX** command), 232
- \dot** (‘math accent), 50  
**\doteq** (≈), 43  
**\dotfill**, 102  
 dotless i and j, 38, 50, 190  
 dots, space-filling, 102  
 double quote, 13, 170  
 double spacing, 172  
 double sub- or superscript error, 140, 141  
 double-column, *see* two-column  
**\doublerulesep**, 207  
**\Downarrow** (⇓), 43, 47  
**\downarrow** (↓), 43, 47  
**draft** document-class option, 93, 177  
**draft** package option, 131  
 drawing curves, 125–126  
 drawing pictures, 118–129  
 driver, device, 118
- dtx** file, 92
- dvi** (device-independent) file, 6, 208  
 page written to, 135
- e-mail  
 sending document by, 77  
 sending file by, 32
- editing, copy, double-spacing for, 172
- edition** field (in **bib** file), 163
- editor** field (in **bib** file), 163
- 11pt** document-class option, 19
- eleven-point type, 19, 115

\ell (ℓ), 43  
ellipsis, 15, 33, 40–41, 189  
em (unit of length), 99, 215  
\em, 27  
\emph (emphasis), 16, 33, 171  
  in slides, 81  
emphasis, 16, 33, 171  
empty \mbox, 97  
empty page style, 89, 179  
  for title page, 90  
\emptyset (Ø), 43  
\encl, 85  
end of line  
  character, 12  
  ending command name with, 16  
  space character at, 19  
end of paragraph, 13, 166, 171  
end of sentence, 13  
end of word, 13  
\end, 23, 167  
  delimits scope of declaration, 27–28,  
  168  
  is fragile, 22, 33  
  TeX command, 233  
  unmatched, 136  
\end{document}, 19, 170  
  error while processing, 135, 143  
  figures and tables output by, 198  
  lof file written by, 208  
  lot file written by, 208  
  missing, 30  
  toc file written by, 209  
\end{verbatim}, no space allowed in, 64  
\end..., defining, 55  
\endinsert (TeX command), 232  
\enlargethispage, 214  
\enlargethispage\*, 96, 214  
\ensuremath, 53, 187  
enter key, 12  
entry  
  bib file, 156  
  field, bibliography, 156, 160–164  
  index, 75  
  type, bibliography, 156, 160–162  
enumerate environment, 24–25, 34, 184  
  as list, 112  
counters, 97  
  suppressing advance of, 185  
cross-reference to item number, 68  
enumerated list, long, 137  
enumi ... enumiv counters, 97, 184  
environment, 23, 33, 167  
  \*-form of, 167  
  \begin of nonexistent, 137  
  blank line above or below, 23  
  defining in terms of a command, 108  
  definition, 55–56, 192–193  
  with optional argument, 193  
invisible, 169  
list-making, 24–25, 112–115  
made from declaration, 27, 33  
math mode, 187–189  
nested too deeply, 144  
paragraph-making, 183–187  
parameter, 56, 193  
redefining, 56  
theorem-like, 56–57, 193–194  
  as displayed paragraph, 183  
  cross-reference to, 68  
undefined, 137  
\epsilon (ε), 41  
\eqalign (TeX command), 233  
\eqalignno (TeX command), 233  
eqnarray environment, 47, 188  
  \\ in, 169  
  anomalous vertical space after, 184  
  cross-reference to equation number, 68  
  formula numbers in, 88  
  in leqno document-class option, 177  
  space between rows in, 188  
  too many columns, 139  
eqnarray\* environment, 48, 188  
\equal, 195  
equation counter, 97, 188  
equation environment, 26, 34, 39, 188  
  anomalous vertical space after, 184  
  cross-reference to number, 68  
  formula numbers in, 88  
  in leqno document-class option, 177  
\equiv (≡), 43  
error, 133–147  
  catching with text editor, 32  
  common, 31

- finding, 134–136  
 in cross-referencing file, 66  
 indicator, 29, 134  
 L<sup>A</sup>T<sub>E</sub>X versus T<sub>E</sub>X, 29  
 locator, 29, 134, 135  
 message, 29–30
  - L<sup>A</sup>T<sub>E</sub>X, 136–140
  - MakeIndex*, 154
  - T<sub>E</sub>X, 140–145
 outputting, 135  
 T<sub>E</sub>X versus L<sup>A</sup>T<sub>E</sub>X, 29  
 typographic, 91  
**errsam.tex** file, 28  
 et al., 158  
 $\backslash\text{eta}(\eta)$ , 41  
 $\backslash\text{evenasidemargin}$ , 181, 182  
**ex** (unit of length), 99, 215  
 exception dictionary, 143  
 exclamation point, *see* !  
 executive paper size, 177  
**executivepaper** document-class option, 177  
 $\backslash\text{exists}(\exists)$ , 43  
 $\backslash\text{exp}(\exp)$ , 44  
 extension, file, 12  
 $\backslash\text{extracolsep}$ , 205  
  
 face example, 108–110  
 $\backslash\text{face}$ , 108–110  
 family of type, 36, *see* type  
 fat, making things, 129  
 $\backslash\text{fbox}$ , 104, 125, 217
  - width of line, 219 $\backslash\text{fboxrule}$ , 219
  - not used in picture commands, 222
  - used by  $\backslash\text{fcolorbox}$ , 225 $\backslash\text{fboxsep}$ , 219
  - used by  $\backslash\text{colorbox}$ , 225 $\backslash\text{fcolorbox}$ , 132, 225  
 field, bibliography entry, 156, 160–164  
 figure, 58–59, 197–200
  - centering, 112
  - in two-column format, 197
  - placement of, *see* float
  - too tall, 145
  - vertical space in, 58
 figure counter, 97  
  
**figure** environment, 58–59, 197
  - in **parbox**, 138
  - misplaced, 138
  - parbox** made by, 104
  - space around, 169
  - too many, 139**figure\*** environment, 197  
 figures, list of, 67, 175–176
  - generated from **lof** file, 208
 file, 207–209
  - ASCII, 144
  - auxiliary, error when reading, 135, 228
  - bibliographic database, *see* **bib** file
  - chapter in separate, 74
  - cross-referencing, 66
  - device-independent, *see* **dvi** file
  - extension, 12
  - $\backslash\text{include}'d$ , 73
  - input, *see* input
  - inserting, 72
  - name, 12
  - needs format error, 139
  - nonexistent, 137, 141
  - not found error, 137
  - of declarations, 73
  - prepended, 170
  - root, 72
  - sample input, 2, 8
  - sending by e-mail, 32
  - text, 12, 144**filecontents** environment, 77, 170, 211  
**filecontents\*** environment, 211  
 files, listing ones used, 77  
 files, multiple input, 72–74  
 $\backslash\text{fill}$ , 100, 102, 215
  - in **tabular\*** environment, 205
 final document-class option, 177  
 final package option, 131  
 finding an error, 134–136  
 first name of file, 12  
 first page, right head for, 180  
 $\backslash\text{fivebf}$  (T<sub>E</sub>X command), 233  
 $\backslash\text{fivei}$  (T<sub>E</sub>X command), 233  
 $\backslash\text{fiverm}$  (T<sub>E</sub>X command), 233  
 $\backslash\text{fivesy}$  (T<sub>E</sub>X command), 233  
 $\backslash\text{flat}(\flat)$ , 43

- fleqn** document-class option, 88, 177, 188, 189  
indentation in, 178, 188
- float**, 58–59, 199  
page, 199  
made by `\clearpage`, 97  
placement of, 59, 197  
specifier, 197  
too tall, 145
- \floatpagefraction**, 200  
floats, too many, 139
- \floatsep**, 200
- floppy disk**, *see* diskette
- flush left**  
array column, 45  
text, 111
- flush right**  
array column, 45  
text, 111  
in **tabbing** environment, 203
- \flushbottom**, 88  
bad page break with, 96  
ignored by `\newpage`, 97  
`\parskip` value for, 172  
space between paragraphs, 100
- flushleft** environment, 111  
`\\\` in, 169  
as displayed paragraph, 183  
as list, 112
- flushright** environment, 111  
`\\\` in, 169  
as displayed paragraph, 183  
as list, 112
- \fnsymbol**, 195
- \folio** (TeX command), 232
- font**, 115–116, 225–226  
Computer Modern, 115  
for slides, 81  
length dependent on, 99  
selecting in Plain TeX, 233  
shape not available, 145, 146  
special, 116  
warning caused by `\boldmath`, 52, 145
- foot**, page, 89, 179
- \footline** (TeX command), 232
- footnote**, 17, 172–173  
colors in, when split across pages, 132
- example of difficult, 173  
in **minipage** environment, 105, 218  
in **parbox**, 105  
line above, 173  
mark, 172  
symbols, 195  
too many, 137  
type size in, 116
- footnote** counter, 97  
for **minipage** environment, 97
- \footnote**, 17, 33, 172  
in **minipage** environment, 105, 218  
is fragile, 22, 33
- \footnotemark**, 172  
for footnote in **parbox**, 105
- \footnoterule**, 173
- \footnotesep**, 173
- \footnotesize**, 115, 226
- \footnotetext**, 173  
for footnote in **parbox**, 105  
in **minipage** environment, 218
- \footskip**, 182
- \footstrut** (TeX command), 232
- \forall** (forall), 43
- foreign language, *see* non-English language
- forests, preserving, 6
- form**, command, 166
- format**  
open bibliography, 177  
this file needs, 139  
two-column, 20, 59, 88, 180, 197
- formatting the input file, 31
- formatting, visual, 64  
for slides, 81
- formula**, math, 18, 33, 39–53, 187–191  
aligning on left, 88, 177  
arrow in, 53  
blank line not allowed in, 142, 171  
bold subformula of, 52  
box in, 103  
center line of, 46  
changing type size in, 116  
complicated, 52  
delimiter in, 190  
displayed, *see* displayed formula

- formatting with `picture` environment, 52  
 lines in, 53  
 logical structure of, 50  
`\mbox` in, 38, 39  
 multiline, 47–49  
 number on left, 177  
 numbered, 39  
 overprinting of number, 188  
 space character in, 18, 50  
 space in, 50–51  
 splitting across lines, 188  
 using `graphics` package for, 53  
 vertical space in, 106  
 visual formatting of, 49, 52
- `\frac`, 40, 189  
 fraction, 40, 189  
 fragile command, 22, 25, 26, 33, 167–168  
 in `\mathbb{C}`-expression, 168  
 in moving argument, 135  
 protecting, 22
- `\frame`, 125, 223  
`\framebox`, 104, 125, 217  
 in `picture` environment, 120, 221  
 use of `\width`, `\height`, `\depth`, and  
`\totalheight` in, 219  
 width of line, 219
- framed box, 104, 217  
 in `picture` environment, 222  
 framing, 125
- `\frenchspacing`, 171  
`From_L`, line beginning with, 32  
`\From`, 32  
 front matter (of a book), 80  
`\frontmatter`, 80  
`\frown` ( $\sim$ ), 43  
 function, log-like, 44–45, 190  
`\fussy`, 95, 214
- galley, 135  
`\Gamma` ( $\Gamma$ ), 41  
`\gamma` ( $\gamma$ ), 41  
`\gcd` ( $\gcd$ ), 44  
 subscript of, 190  
 geometric transformation, 129  
 geometry, coordinate, 118  
`\geq` ( $\geq$ ), 43
- `\gg` ( $\gg$ ), 43  
 Gilkerson, Ellen, iii, xv, 130  
`\glo` (glossary) file, 75, 208  
 suppressed by `\nofiles`, 212  
 written by `\makeglossary`, 212
- global declaration, 168  
 global option, unused, 147  
 glossary, 74–76, 211–212  
`\glossary`, 75, 212  
`\glo` file entry written by, 208  
 space around, 169  
 too many on page, 143
- `\glossaryentry`, 75, 208, 212  
 gnomonly, 93  
 gnu, 37  
 Goossens, Michel, xv, 2  
 Gordon, Peter, xvi  
 graphic, determining size of, 137  
`\graphics` package, 129–131, 178, 223–224  
 making figures with, 58  
 making formulas with, 53
- graphics, unknown extension, 140  
`\graphpap` package, 126, 178, 221  
`\graphpaper`, 126–127, 221  
`\grave` ( $\acute{}$  math accent), 50  
 gray color model, 132, 225  
 Greek letter, 41  
 green, 132, 225  
 grid, coordinate, 126–127
- `\h` float specifier, 197  
 changed, 146  
`\H` ( $\hat{}$  accent), 38  
 half oval, 124  
 hash size, 143  
`\hat` ( $\hat{}$  math accent), 50  
`\hbar` ( $\hbar$ ), 43  
`\hbox`, overfull, 30, 93, 147  
 marked with `\draft` option, 93  
`\hbox`, underfull, 95, 147  
 head, of arrow, 123  
 head, page, 89, 179  
 in two-sided printing, 89  
 set by sectioning command, 22, 90, 174  
`\headheight`, 182

- heading, *see* head  
**headings** page style, 89, 179, 180  
**\headline** (TeX command), 232  
**\headsep**, 182  
**\heartsuit** (♡), 43  
**height**  
    changing apparent, 107  
    of a box, 103  
    of page body, 100, 214  
**\height**, 219  
Helvetica, *see* sans serif type family  
hexadecimal character code, 116  
**\hfill**, 102, 217  
    in marginal note, 59  
    positioning item label with, 114  
    used with **\vline**, 206  
**\hline**, 62, 206  
**\hom** (hom), 44  
**\hookleftarrow** (←), 43  
**\hookrightarrow** (→), 43  
horizontal  
    brace, 49  
    line  
        drawn with **\rule**, 106  
        in **array** or **tabular** environment, 62, 206  
        space-filling, 102  
    mode, 36  
    positioning of text, 121  
space, 101  
    ;  
    around **array** or **tabular**  
        environment, 205  
    in formula, 51, 191  
**howpublished** field (in **bib** file), 163  
**\rulefill**, 102  
**\hspace**, 101, 216  
    rubber length in, 102  
**\hspace\***, 102, 216  
**\Huge**, 115, 226  
**\huge**, 115, 226  
hyphen, 14.  
    inserted by TeX, 17  
**hyphenation**, 93  
    correcting error in, 94  
    of non-English words, 94  
    permitting with **\-**, 213  
    suppressed in typewriter style, 226  
**\hyphenation**, 94, 213  
     $\backslash-$  instead of, 143  
    error in, 142  
    exceeding capacity with, 143  
    scope of, 168  
**i**, dotless (for accents), 38, 43, 50, 190  
**\i** (i), 38  
**\idx** (index) file, 74, 150, 208  
    suppressed by **\nofiles**, 212  
    written by **\makeindex**, 212  
**\ifthen** package, 178, 195–196  
**\ifthenelse**, 195  
ignored bibliography field, 161  
ignoring input, 19  
illegal  
    character in array argument, 137  
    parameter number, 141  
    unit of measure, 141  
**\Im** (ʒ), 43  
**\imath** (i), 43, 50, 190  
**\in** (inch), 99, 215  
**\in** (€), 43  
in-text formula, 39  
**inbook** bibliography entry type, 161  
**inch** (in), 99, 215  
**\include**, 73, 211  
    cannot be nested, 137  
    numbering with, 74  
    of nonexistent file, 137  
    sending files read by, 77  
    uses **\clearpage**, 97  
**\include'd** file, 73  
    **\newcounter** in, 138  
**\includegraphics**, 130, 224  
    cannot determine size of error, 137  
**\includegraphics\***, 224  
**\includeonly**, 73, 211  
    entered from terminal, 76  
    entering arguments with **\typein**, 74  
    misplaced, 136  
**incollection** bibliography entry type, 161  
**\ind** file, 150, 208  
    read by **\printindex**, 211  
**\indent**, 171  
indentation, in **fleqn** option, 178

- indentation, paragraph, *see* paragraph  
 index, 30, 74–76, 150–154, 211–212  
`\index`, 74, 212
  - curly brace in argument, 75
  - `idx` file entry written by, 208
  - in command argument, 153
  - space around, 169
  - special character in argument, 75
  - too many on page, 143`\indexentry`, 75, 212
  - on `idx` file, 208`\indexspace`, 75  
 indicator, error, 29, 134  
`\inf` (`inf`), 44
  - subscript of, 190
 infinite loop, 252  
 infinitely stretchable length, 100, 102, 215  
 information, moving, 65–77, 207–209  
`\infty` ( $\infty$ ), 43  
 ink, rectangular blob of, 106  
`\inproceedings` bibliography entry type, 161  
 input
  - character, 32
  - file, 12
    - displaying logical structure, 53
    - formatting, 31
    - page of, 13
  - files, multiple, 72–74
    - finding error in, 134
  - from terminal, 76–77, 212–213
  - ignoring, 19
  - processing part of, 73–74
  - sample, 2, 8`\input`, 72, 210
  - braces missing from argument, 141
  - differs from Plain TeX version, 233
  - of nonexistent file, 137
  - sending files read by, 77
 inserting a file, 72  
`\institution` field (in `bib` file), 163  
`\int` ( $\int$ ), 44, 51  
 integral sign, space around, 51  
 interaction, 76–77  
 intercolumn space, 178
  - in `array` or `tabular` environment, 205
 interrow space
  - in `array` or `tabular` environment, 169, 207
  - in `eqnarray` environment, 188
  - in `\shortstack`, 124
 interword space, 14, 170
  - before or after `\hspace` command, 101
  - in math mode, 51
  - produced by invisible command, 169
  - too much with `\sloppy`, 214
  - without line break, 17, 170`\intertsep`, 200  
 intraword dash, 14, 170  
 invalid character error, 144  
 invisible
  - character, 12, 32
  - command, 169
  - delimiter, 47
  - environment, 169
  - term made with `\mbox`, 48
  - text, 82, 97`\iota` ( $\iota$ ), 41  
`\isodd`, 196  
 italic type shape, 16, 37, 226
  - in math mode, 51, 191
  - used for emphasis, 171`\item`, 24–25, 34, 184
  - [ following, 25
  - in index, 211
  - in `theindex` environment, 75
  - in `trivlist` environment, 115
  - is fragile, 33
  - lonely, 137
  - missing, 138
  - optional argument of, 25, 167
  - outside list environment, 137
  - `\ref` value set by, 209
 item, label, 24–25, 184
  - extra-wide, 114
  - positioning with `\hfill`, 114`\itemindent`, 113, 186
  - in `trivlist` environment, 115`\itemize` environment, 24–25, 34, 184
  - as list, 112
  - default labels of, 184`\itemsep`, 113, 185  
`\itshape`, 37, 226

- j, dotless (for accents), 38, 43, 50, 190  
\j (j), 38  
\jmath (j), 43, 50, 190  
\Join (ℳ), 43  
\jot, 188  
journal field (in **bib** file), 163  
Jr., 157  
justifying lines, 95, 100
- \kappa (κ), 41  
\ker (ker), 44  
Kernighan, Brian, 7  
key  
  citation, 69, 71, 156  
  cross-reference, 67  
  enter, 12  
  return, 12  
key field (in **bib** file), 162, 163  
keyboard, *see* terminal  
keys, listing, 69  
\kill, 61, 202  
Knuth, Donald Ervin, xvi, 5, 115
- l (left)  
  oval-part argument, 124, 223  
  positioning argument, 45, 217, 221  
    in **array** or **tabular** argument, 205  
    of **\makebox**, 104, 121  
    of **\shortstack**, 124  
l (letter el), 12  
\L (L), 39  
\l (l), 39  
label  
  item, 24–25, 184  
  extra-wide, 114  
  positioning with **\hfill**, 114  
mailing, 86  
multiply-defined, 146  
produced by **\cite**, 210  
source, 69  
\label, 67, 209  
  in **\caption** argument, 209  
  in **eqnarray** environment, 188  
  in **figure** or **table** environment, 68  
missing, 146  
preceded by **\caption**, 68  
similar to **\bibitem**, 71
- space around, 169  
**labelitemi** ... **labelitemiv** counters, 184  
labels may have changed warning, 146  
labels, cross-reference, too many, 143, 144  
\labelsep, 113, 186  
\labelwidth, 113, 186  
  in **trivlist** environment, 115  
**lablst.tex** file, 69, 208  
\Lambda (Λ), 41  
\lambda (λ), 41  
Lamport, Jason, 131  
Lamport, Leslie, 131  
**landscape** document-class option, 177  
landscape printing, 177  
\langle (⟨), 47  
language, non-English, 38, 94  
\LARGE, 115, 226  
\Large, 115, 226  
\large, 115, 226  
**L<sup>A</sup>T<sub>E</sub>X**  
  bug, 139  
  distinguished from **T<sub>E</sub>X**, 5  
  error message, 136–140  
  error versus **T<sub>E</sub>X** error, 29  
  logo, 5, 15, 33, 171  
  pronunciation of, 5  
  running on part of document, 31, 73–74  
  running unattended, 30  
  stopping, 30, 136  
  version, 2  
  warning message, 135, 145–147  
**L<sup>A</sup>T<sub>E</sub>X Companion**, The, 2  
\LaTeX, 15, 33, 171  
**L<sup>A</sup>T<sub>E</sub>X 2.09** versus **L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>**, 2, 228–229  
**L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>**, xv, 2, 118, 132  
**latextsym** package, 42, 178  
law (mathematical), 56  
\lceil (⌈), 47  
\ldots, 15, 33  
\ldots (..., 40, 189  
\leadsto (⇝), 43  
leaving math mode, 36  
left margin, prevailing, 201  
left\_margin\_tab, 201

**\left**, 47, 190  
**left**, *flush*, *see* *flush*  
**left-hand page**, 179  
**left-to-right mode**, *see* *LR mode*  
**\Leftarrow** ( $\Leftarrow$ ), 43  
**\leftarrow** ( $\leftarrow$ ), 43  
**\lefteqn**, 48, 188  
**\lefteye**, 109  
**\leftharpoondown** ( $\leftharpoondown$ ), 43  
**\leftharpoonup** ( $\leftharpoonup$ ), 43  
**\leftmargin**, 113, 185  
  *in trivlist environment*, 115  
**\leftmargini** ... **\leftmarginvi**, 185  
**\Leftrightarrow** ( $\Leftrightarrow$ ), 43  
**\rightleftarrows** ( $\rightleftarrows$ ), 43  
**legal paper size**, 177  
**legalpaper** document-class option, 177  
**lemma**, 56  
**length**, 99–101, 215–216  
  *command*, 99  
  *defining*, 101  
  *not preceded by \protect*, 168  
**font-dependent**, 99  
**infinitely stretchable**, 100, 215  
**natural**, 100  
**of line in picture**, 122  
**parameter**, 100  
**rigid**, 100  
**rubber**, 100  
  *in \hspace or \vspace*, 102  
  *in \lengthtest expression*, 196  
  *infinitely stretchable*, 102  
**unit (in picture environment)**, 118  
**unit of**, 215  
**zero**, 99  
**\lengthtest**, 196  
**\leq** ( $\leq$ ), 43  
**\leqalignno** (TeX command), 233  
**leqno** document-class option, 88, 177,  
  188  
**letter**, 32  
  *case of*, 68, 69  
  *for mailing*, 84–86  
  *Greek*, 41  
  *lowercase*, 12  
  *not a*, 142  
  *script*, 42  
**uppercase**, 12  
**letter** document class, 84–86, 176  
  *\parskip in*, 100  
**letter environment**  
  *moving argument of*, 168  
**letter environment**, 85  
**letter paper size**, 177  
**letterpaper** document-class option, 177  
**level number of sectional unit**, 176  
**\lfloor** ( $\lfloor$ ), 47  
**\lg** ( $\lg$ ), 44  
**\lhd** ( $\lhd$ ), 42  
**\lim** ( $\lim$ ), 44  
  *subscript of*, 190  
**\liminf** ( $\liminf$ ), 44  
  *subscript of*, 190  
**\limsup** ( $\limsup$ ), 44  
  *subscript of*, 190  
**line**  
  *blank*, *see* *blank line*  
  *bottom of*, 107  
  *break*, *see* *break, line*  
  *horizontal*, *see* *horizontal*  
  *in formula*, 53  
  *in picture*, 122–123  
  *thickness of*, 118  
  *justifying*, 95, 100  
  *none to end error*, 139  
  *slanted*, *minimum size of*, 123  
  *space at beginning or end of*, 102  
  *vertical*, *see* *vertical*  
  *width*, *see* *width*  
**\line**, 122, 222  
  *error in*, 136  
  *TeX command*, 233  
**\linebreak**, 95, 213  
  *optional argument of*, 167  
  *space around*, 169  
  *warning caused by*, 147  
**lines**, *distance between*, 100  
**\linethickness**, 223  
**\linewidth**, 171  
**lining up**, *see* *aligning*  
**list**, 24–25, 34  
  *counter, defining*, 114  
  *long enumerated*, 137  
  *margins of nested*, 114

- of figures or tables, 67, 175–176, 208  
adding an entry, 175  
adding commands to, 67, 176  
error in, 135  
source, 70
- list** environment, 112–114, 185  
style parameters for, 185
- list-making environment, 24–25, 112–115  
defining, 114  
in **parbox**, 105, 218  
nested too deeply, 139  
primitive, 112
- \listfiles**, 77, 211
- listing keys, 69
- \listoffigures**, 67, 175–176  
error when processing, 143  
**lof** file read by, 208
- \listoftables**, 67, 175–176  
error when processing, 143  
**lot** file read by, 208
- \listparindent**, 113, 185
- \ll** (⟨⟨), 43
- \ln** (ln), 44
- local coordinates, 129
- Local Guide, 2, 6, 8, 12, 20, 28, 30, 69, 71, 77, 86, 88, 91–93, 115, 116, 118, 124, 132, 136, 139, 140, 151, 156, 159, 163, 178
- locator, error, 29, 134, 135
- lof** (list of figures) file, 67, 175, 208  
editing, 67  
error in, 135
- log** file, 28, 76, 208
- \log** (log), 44, 51
- log-like function, 44–45, 190
- logical design, 7
- logical structure, 6, 88  
displaying in input file, 53  
of formula, 50  
repeated, 53
- logo, **LATEX**, 5, 15, 33, 171
- logo, **TEX**, 5, 15, 33, 171
- lonely **\item**, 137
- \Longleftarrow** (⟨⟨), 43
- \longleftarrow** (⟨⟨), 43
- \Longleftrightarrow** (⟨⟨), 43
- \longleftrightarrow** (⟨⟨), 43
- \longmapsto** (→→), 43
- \Longrightarrow** (→→), 43
- \longrightarrow** (→→), 43
- lot** (list of tables) file, 67, 175, 208  
editing, 67  
error in, 135
- low ellipsis, 40, 189
- lowercase letter, 12
- lowering text, 107
- LR box, 103–104, 107
- LR mode, 36, 39  
in **tabbing** environment, 201  
space character in, 36  
**tabular** item processed in, 62
- lrbox** environment, 108, 218
- macho **TEX** programmer, 92
- macro parameter character error, 145
- magenta, 132, 225
- magnification of output, 233
- \magnification** (**TEX** command), 233
- magnifying a picture, 118
- Magritte, René, 131
- mailing label, 86
- main matter (of a book), 80
- main memory size, 143
- \mainmatter**, 80
- \makebox**, 104, 217  
in **picture** environment, 120, 221  
use of **\width**, **\height**, **\depth**, and  
**\totalheight** in, 219
- \makefootline** (**TEX** command), 232
- \makeglossary**, 75, 212  
**glo** file produced by, 208
- \makeheadline** (**TEX** command), 232
- makeidx** package, 150  
defines **\printindex**, 211
- MakeIndex**, 74, 150–154  
**ind** file written by, 208
- \makeindex**, 74, 150, 212  
**idx** file produced by, 208  
misplaced, 136
- \makelabel**, 186
- \makelabels**, 86
- \maketitle**, 20, 34, 90, 181  
effect of **titlepage** option, 88  
not preceded by **\author**, 146

- not preceded by `\title`, 138  
 mandatory argument, 166  
`manual` bibliography entry type, 162  
`\mapsto` ( $\rightarrow$ ), 43  
 margin  
     arrow in, 59  
     changing in `tabbing` environment, 202  
     determined by `\textwidth` and  
         `\textheight`, 100  
     of nested lists, 114  
     prevailing, 201, 203  
 marginal note, 59–60, 200–201  
     `\hfill` in, 59  
     moved, 146  
     overprinting of, 201  
 marginpar moved warning, 59, 146  
`\marginpar`, 59–60, 200  
     incorrectly placed, 146  
     space around, 169  
     too many on page, 139  
`\marginparpush`, 201  
`\marginparsep`, 182, 201  
`\marginparwidth`, 182, 201  
 mark, footnote, 172  
`\markboth`, 89, 90, 179  
     moving argument of, 90, 168  
     with `myheadings` page style, 179  
`\markright`, 89, 90, 179  
     moving argument of, 90, 168  
     with `myheadings` page style, 179  
`mastersthesis` bibliography entry type, 162  
 math  
     formula, *see* formula, math  
     mode, 36, 39  
         accent in, *see* accent  
         bad command in, 137, 145  
         blank line not allowed in, 142, 171  
         defining commands for use in, 53  
         environment, 187–189  
         leaving, 36  
         space character ignored in, 36, 50  
     style, 52, 191  
         display, 52, 188, 191  
         for sub- and superscripts, 52, 191  
         of `array` environment item, 191  
         text, 52, 187, 191
- symbol, 41–45  
     variable-sized, 42, 52  
`math` environment, 18, 39, 187  
`\mathbf`, 51, 191  
`\mathcal`, 42, 51, 191  
 mathematical, *see* math  
`\mathindent`, 178, 188  
`\mathit`, 51, 191  
`\mathrm`, 51, 191  
`\mathsf`, 51, 191  
`\mathtt`, 51, 191  
 matrix, 45  
`\max` (`max`), 44  
     subscript of, 190  
`\mbox`, 17, 33, 104, 171, 217  
     bold subformula in, 52  
     empty, 97  
     for changing type size in formula, 116  
     how it works, 36, 103  
     in formula, 38, 39  
     invisible term made with, 48  
`\mdseries`, 37, 226  
 medium space, 51, 191  
 medium type series, 37, 226  
`\medskip`, 216  
`\medskipamount`, 216  
 memory size, 143  
 message  
      $\text{\LaTeX}$  error, 136–140  
      $\text{\LaTeX}$  warning, 145–147  
         page number in, 135  
     `MakeIndex` error, 154  
     printing on terminal, 76  
      $\text{\TeX}$  error, 140–145  
      $\text{\TeX}$  warning, 30, 147  
`\mho` ( $\mathcal{U}$ ), 43  
`\mid` ( $|$ ), 43  
`\midinsert` ( $\text{\TeX}$  command), 232  
 millimeter (`mm`), 99, 215  
`\min` (`min`), 44  
     subscript of, 190  
`\minipage` environment, 104, 105, 218  
     `footnote` counter for, 97  
     footnote in, 172  
     in p column of `array` or `tabular`  
         environment, 205  
     nested, 106

- tabbing environment in, 106  
versus `\parbox`, 105
- minus sign, 14
- mirror image, 130
- misc bibliography entry type, 162
- misplaced
- `#`, 141, 145
  - `&`, 141
  - alignment tab, 141
  - figure environment, 138
  - `\includeonly`, 136
  - `\makeindex`, 136
  - `\marginpar`, 138
  - `\nofiles`, 136
  - table environment, 138
  - `\usepackage`, 136
- missing
- `$` error, 142
  - `\``, 141
  - `{` error, 142
  - `}` error, 142
  - `\texttt{O}`-expression, 138
  - argument, 138, 142
  - `\begin{document}` error, 138
  - brace, 142
  - control sequence error, 142
  - `\documentclass`, 144
  - `\end{document}`, 30
  - `\item`, 138
  - `\label`, 146
  - number error, 142
  - `\parg` error, 138
  - `\usepackage`, 144
- misspelled command name, 31
- correcting, 144
- Mittelbach, Frank, xv, 2
- `mm` (millimeter), 99, 215
- mod, 44, 190
- mode, 36
- compatibility, 228
  - horizontal, 36
  - left-to-right, *see* LR mode
  - LR, *see* LR mode
  - math, *see* math
  - paragraph, *see* paragraph
  - picture, 120, 220–221
  - vertical, 36
- model, color, 132, 224  
undefined, 140
- `\models (\models)`, 43
- modulo, 44, 190
- month field (in `bib` file), 163
- moved marginal note, 146
- moving argument, 22, 33, 167
- fragile command in, 135
  - of `\texttt{O}`-expression, 205
  - of `\bibitem`, 72
  - of `\caption`, 58
  - of `\letter` environment, 85
  - of `\markboth` and `\markright`, 90
  - of `\typein` and `\typeout`, 77, 212, 213
- moving information around, 65–77, 207–209
- `\mp (\mp)`, 42
- `\mpfootnote` counter, 97
- `\mu (\mu)`, 41
- `\multicolumn`, 62, 206
- error in, 137, 138
  - not allowed in `eqnarray`, 188
- multilanguage document, 38
- multiline formula, 47–49
- multiple
- authors, 21
  - captions, 59
  - column item, 62, 206
  - input files, 72–74
    - finding error in, 134
    - names in `bib` file field, 158
- multiply-defined label warning, 146
- `\multiput`, 127–128, 221
- `\myheadings` page style, 89, 179
- `\nabla (\nabla)`, 43
- name
- command, *see* command
  - in `bib` file field, 157–158
    - of file, 12
- named theorem, 57
- names, multiple, in `bib` file field, 158
- natural length, 100
- `\natural (\natural)`, 43
- `\nearrow (\nearrow)`, 43
- `\neg (\neg)`, 43
- negative thin space, 51, 191

- \neq ( $\neq$ ), 43
- nested
  - commands, 16
  - lists, margins of, 114
  - `minipage` environments, 106
  - too deeply, 139, 144
- nesting depth error, 139
- \newboolean, 196
- \newcommand, 53, 168, 192
  - braces removed from, 92
  - error in, 137, 141, 142
- \newcounter, 99, 194
  - error in, 137, 138
  - in included file, 138
  - optional argument of, 167
  - scope of, 168
- \newenvironment, 55–56, 114, 192
  - error in, 137, 141, 144
- \newlength, 101, 216
  - error in, 137, 142
  - scope of, 168
- \newline, 95, 213
  - bad use of, 147
  - error in, 139
- \newpage, 97, 215
  - in two-column format, 97
- \newsavebox, 107, 218
  - error in, 137, 142
  - scope of, 168
- newt, 68
- \newtheorem, 56, 193
  - counter created by, 97
  - cross-reference to environment defined by, 68
  - environment defined by, 183
  - error in, 137, 138, 144
  - optional argument of, 167
  - scope of, 168
- next\\_tab\\_stop, 201
- \ni ( $\ni$ ), 43
- Nixon, Richard, 24
- no counter error, 138
- \nocite, 70, 210
- \nofiles, 207
  - misplaced, 136
  - suppresses `glo` file, 212
  - suppresses `idx` file, 212
- used when editing `toc` file, 67
- \noindent, 171, 183
- \nolinebreak, 95, 213
  - optional argument of, 167
  - space around, 169
- non-English symbol, 38–39
  - in `bib` file, 158
- nonexistent
  - document class, 137
  - environment, 137
  - file, 141
  - package, 137
- \nonfrenchspacing, 171
- nonmath symbol, 38–39
- nonprinting character, 144
- \nonumber, 48, 188
- \nopagebreak, 96, 214
  - optional argument of, 167
  - space around, 169
- \nopagenumbers (`TEX` command), 232
- \normalbottom (`TEX` command), 232
- \normalfont, 226
- \normalmarginpar, 201
- \normalsize, 115, 226
- \not, 42
  - in `ifthen` package expression, 196
- notation, 53
  - versus concepts, 31
- \note environment, 83
- \note field (in `bib` file), 163
- note, marginal, 59–60, 200–201
- \notin ( $\notin$ ), 43
- \notitlepage document-class option, 177
- \nu ( $\nu$ ), 41
- number
  - alphabetic, 195
  - arabic, 195
  - assigning key to, 67
  - formula, 88
    - printed at left, 177
    - suppressing in `eqnarray`, 48
  - illegal parameter, 141
  - missing, 142
  - page, *see* page
  - roman, 195
  - wrong, 146
- \number field (in `bib` file), 163

number-range dash, 14, 170  
numbered displayed formula, 39  
numbering, 97–99, 194–195  
  commands, `\the...`, 98  
  page, 98  
  sectional units, 176  
  style, 98  
  with `\include`, 74  
  within sectional unit, 57  
`\numberline`, 176  
`\narrowarrow{}`, 43

`0` (letter oh), 12  
`\O` ( $\emptyset$ ), 39  
`\o` ( $\emptyset$ ), 39  
object, floating, 58  
object, picture, 120, 221–223  
octal character code, 116  
`\oddsidemargin`, 181, 182  
`\odot` ( $\odot$ ), 42  
`\OE` ( $\text{OE}$ ), 39  
`\oe` ( $\text{oe}$ ), 39  
`\oint` ( $\oint$ ), 44  
`\oldstyle` (TeX command), 233  
`\Omega` ( $\Omega$ ), 41  
`\omega` ( $\omega$ ), 41  
`\omicron` ( $\circ$ ), 41  
`\ominus` ( $\ominus$ ), 42  
omitted argument, error caused by, 141  
one (1), 12  
one-column format, 181  
one-sided printing, marginal notes in, 59  
`\onecolumn` document-class option, 177  
`\onecolumn`, 88, 181  
`\oneside` document-class option, 177  
only in preamble error, 136  
`\onlynotes`, 84  
`\onlyslides`, 83  
open bibliography format, 71, 177  
`\openany` document-class option, 80, 177  
`\openbib` document-class option, 71, 177  
`\opening`, 85  
`\openright` document-class option, 80,  
  177  
`\oplus` ( $\oplus$ ), 42  
option  
  clash error, 138  
  document-class, 176  
  document-style (L<sup>A</sup>T<sub>E</sub>X 2.09), 228  
  unknown, 140  
optional argument, 20, 166  
  [ or ] in, 25  
  default, 166  
  defining a command with, 192  
  defining an environment with, 193  
  of `array` environment, 46  
  of `\item`, 25  
  of `\marginpar`, 59  
  of sectioning command, 174  
  of `\twocolumn` too tall, 146  
  square bracket mistaken for, 142  
  square brackets enclosing, 20, 33, 166  
optional arguments, command with two,  
  166  
optional bibliography field, 160  
`\or`, 196  
`organization` field (in `bib` file), 163  
origin, 118  
`\oslash` ( $\oslash$ ), 42  
`\otimes` ( $\otimes$ ), 42  
outerpar mode, not in, 138  
output  
  line, space at beginning or end of, 102  
  printing, 6, 208  
  routine, Plain TeX, 232  
  to terminal, 76–77, 212–213  
`<output>` printed on terminal, 135  
`\output` routine, 232  
outputting error, 135  
oval, 124–125, 223  
  too small, 146  
`\oval`, 124–125, 223  
oval-part argument, 124  
`\overbrace`, 49  
overfull `\hbox`  
  marked with `draft` option, 93  
  message, 30, 93, 147  
overfull `\vbox` message, 147  
`\overlay` environment, 82  
`\overline`, 49, 190  
overlining, 49, 190  
overprinting  
  of equation number, 188  
  of marginal notes, 201

- overriding item position in **tabular** environment, 63
- p**
  - float specifier, 197
  - in **array** or **tabular** argument, 205
  - \P** (¶), 39, 173
  - p-arg missing error, 138
  - package, 2, 20, 178–179
    - creating your own, 91
    - loaded twice, 138
    - nonexistent, 137
  - page
    - blank, 97
    - made by **\cleardoublepage**, 215
    - with **titlepage** environment, 180
  - body, 89, 179
    - height of, 100, 214
  - break, *see* break, page
  - color of, 132
  - first, right head for, 180
  - float, 199
    - made by **\clearpage**, 97
  - foot, 89, 179
  - head, 89, 179
    - in **twoside** option, 90
    - set by sectioning command, 90, 174
  - input file, 13
  - last, output by **\stop**, 136
  - left-hand, 179, 196
  - number, 98
    - alphabetic, 180
    - arabic, 89, 180
    - cross-reference to, 68
    - in warning message, 135
    - indexing different styles, 154
    - printed on terminal, 135
    - roman, 89, 180
    - style of, 180
  - one-column, 181
  - positioning relative to, 111
  - range, in index, 152
  - right-hand, 179, 196
    - starting on, 97
  - space at top or bottom, 102
  - squeezing extra text on, 96, 214
  - starting a new, 215
- style, 89–90, 179–182
  - default, 89, 179
- title, *see* title
- too many **\index** or **\glossary** commands on, 143
- too many **\marginpar** commands on, 139
- two-column, 20, 88, 180
  - width of text on, 100
- page counter, 97, 98
  - current value of, 196
  - set by **\pagenumbering**, 180
- \pagebody** (TeX command), 232
- \pagebreak**, 96, 214
  - in two-column format, 97
  - optional argument of, 167
  - space around, 169
- \pagecolor**, 132, 225
  - scope of, 168
- \pagecontents** (TeX command), 232
- \pageheight**, 182
- \pageinsert** (TeX command), 232
- \pageno** (TeX command), 232
- \pagenumbering**, 89, 180
  - scope of, 168
- \pageref**, 68, 209
  - ~ used with, 68
  - undefined, 146
  - used with **\isodd**, 196
  - wrong number printed by, 146
- pages field (in bib file), 163
- pages, how TeX makes, 135
- \pagestyle**, 89, 179
  - after **\chapter**, 89
  - scope of, 89
- \pagewidth**, 182
- paper size, 177
- \par**, 171
- paragraph, 13, 171
  - bad end of, 142
  - beginning of, 183
  - displayed, 183–187
  - end of, 13, 166, 171
  - in picture, 104
  - in table item, 104
  - indentation, 171
    - anomalous, 183

- removing with `\noindent`, 183  
width of, 99, 172
- mode, 36  
  `\`` in, 96  
  box made in, 103  
  box typeset in, 104  
  `center` environment in, 111  
  figure or table body processed in, 58  
  marginal note processed in, 59
- new, 33  
unit, 94
- paragraph counter, 97
- `\paragraph`, 21, 174
- paragraph-making environment, 105, 183–187
- paragraphs  
  `\`` between, 139  
  space between, 100, 172
- `\parallel (||)`, 43
- parameter  
  length, 100  
  number error, 141  
  of command, 54, 192  
  of environment, 56, 193  
  style, 166
- parbox, 104–106, 218  
  `\caption` in, 199  
  figure or table environment in, 138  
  in array or tabular column, 205  
  in tabbing environment, 203  
  list-making environment in, 105, 218  
  marginal note typeset in, 200  
  paragraph-making environment in, 105  
  `\parindent` set to zero in, 172  
  positioning with `\raisebox`, 105  
  tabbing environment in, 105  
  tabular environment in, 218
- `\parbox`, 104, 218  
  versus `minipage` environment, 105
- parenthesis, 12, 15
- `\parindent`, 99, 100, 172  
  equals zero in parbox, 105  
  in list environment, 114
- `\parsep`, 113, 185
- `\parskip`, 100, 172  
  in letter document class, 100
- in list environment, 114, 185
- part counter, 97
- part of input, processing, 31, 73–74
- `\part`, 21, 22, 174
- `\partial` ( $\partial$ ), 43
- `\partopsep`, 113, 185  
  when added, 114
- pasting, 58
- pattern, repeated, 127–128
- pc (pica), 215
- period, 33  
  space after, 14–15
- `\perp` ( $\perp$ ), 43
- `\phdthesis` bibliography entry type, 162
- `\Phi` ( $\Phi$ ), 41
- `\phi` ( $\phi$ ), 41
- `\Pi` ( $\Pi$ ), 41
- `\pi` ( $\pi$ ), 41
- pica (pc), 215
- `\pict2e` package, 118, 179, 221–223
- picture, 118–129  
  command, argument of, 119  
  declaration, 223  
  in float, 58  
  line thickness in, 118  
  magnifying, 118  
  mode, 120, 220–221  
  object, 120, 221–223  
  paragraph in, 104  
  reducing, 118
- picture environment, 118–129, 219–223  
  box made by, 103  
  example, 220  
  formatting formula with, 52  
  large, 143  
  making figures with, 58  
  reusing, 107  
  zero-width box in, 121
- placement  
  of figures and tables, 59, 197  
  of marginal note, 59–60  
  of `\protect`, incorrect, 142  
  of tabular environment, 63
- plain bibliography style, 70
- plain page style, 89, 179
- Plain TeX, 231–233
- `\plainoutput` (TeX command), 232

- \pm** ( $\pm$ ), 42  
**\pmod**, 44, 190  
 poetry, 25–26, 34, 184  
 point (unit of length), 93, 99, 215  
 point, reference, 103, 221  
 pool size, 144  
**\poptabs**, 203  
     unmatched, 138  
 position, specifying by coordinate, 118  
 positioning  
     argument, 45–46, 104–105, 121, 124  
     of item label, 114  
     of parbox with **\raisebox**, 105  
     of text, 121, 217, 219  
     relative to a fixed point on page, 111  
     relative to a line of text, 110  
     text in picture, 121  
     vertical, *see* vertical  
**\pounds** (£), 39, 173  
**\Pr** (Pr), 44  
     subscript of, 190  
 preamble, 19, 34, 170  
     command definition in, 55  
     error in, 138  
**\hyphenation** command in, 94  
**\makeindex** in, 74  
     only in error, 136  
     visual design commands in, 88  
**\prec** ( $\prec$ ), 43  
**\preceq** ( $\preceq$ ), 43  
 preloaded command, 91  
 preparing input file, 12  
 prepended file, 170  
 previewer, screen, 6  
 prime symbol, 18, 189  
**\prime** ( $\prime$ ), 43  
 primitive list-making environment, 112  
 principle, 56  
**\printindex**, 150, 211  
     ind file read by, 208  
 printing  
     aux file information, 208  
     counter values, 98  
     landscape, 177  
      $\text{\LaTeX}$  output, 6, 208  
     one-column, 181  
     one-sided, marginal notes in, 59  
     slides and notes separately, 83  
     two-column, 20, 180  
     two-sided, 19, 177  
         increasing page height in, 214  
         marginal notes in, 59  
**proceedings** bibliography entry type, 162  
**\prod** ( $\prod$ ), 44  
 program, formatting a, 60  
 programming in  $\text{\LaTeX}$ , 195  
 pronunciation of  $\text{\LaTeX}$  and  $\text{\TeX}$ , 5  
 proposition, 56  
**\proto** ( $\propto$ ), 43  
**\protect**, 22, 33, 167  
     in  $\mathbb{C}$ -expression, 205  
     in **\caption** argument, 58  
     in **\index** argument, 153  
     in **\typeout** argument, 212  
     incorrect placement of, 142  
     not before length command, 100, 168  
     not in **\addtocounter** or **\setcounter**  
         argument, 168  
     not used with **\value**, 194  
 protecting a fragile command, 22  
**\providemode**, 168, 192  
     error in, 141  
**\ps**, 86  
**\Psi** ( $\Psi$ ), 41  
**\psi** ( $\psi$ ), 41  
**pt** (point), 99, 215  
**publisher** field (in **bib** file), 163  
 punctuation character, 12, 32  
     in key, 68, 69  
 punctuation dash, 14, 170  
 punctuation, space after, 14–15, 170, 171  
**\pushtabs**, 203  
     unmatched, 138  
**\put**, 120, 221  
     sequence replaced by **\multiput**, 127  
     space in argument, 129  
**\qbezier**, 125–126, 221  
      $\text{\TeX}$  space used by, 143  
**\qbeziermax**, 126, 221  
 quarter oval, 124  
 question mark (?), *see* ?

- quotation environment, 24, 34, 184  
as list, 112
- quotation marks, 13–14, 33, 170
- quotation, displayed, 23, 34
- quote  
double, 13, 170  
left ('), 12, 13, 32, 33, 170  
right ('), 12, 13, 15, 32, 33  
in formula, 18  
single, 13, 170
- quote** environment, 23–24, 34, 184  
as list, 112
- quotient symbol (/), 51
- quoting character in index entry, 154
- r** (right)  
oval-part argument, 124, 223  
positioning argument, 45, 217, 221  
in **array** or **tabular** argument, 205  
of **makebox**, 104, 121  
of **shortstack**, 124
- ragged right, 111
- \raggedbottom**, 88  
bad page break with, 96
- \raggedleft**, 112  
in p column, 205
- \raggedright**, 112  
in p column, 205
- \raisebox**, 107, 219  
positioning parbox with, 105  
use of optional arguments, 109  
use of **\width**, **\height**, **\depth**, and  
  **\totalheight** in, 219
- raising text, 107
- \rangle** (⟨), 47
- \rceil** (⟨), 47
- \Re** (⟨), 43
- reading auxiliary file, error when, 135,  
  228
- reclaiming saved box's space, 127
- rectangular blob of ink, 106
- recursive definition, 54
- red, 132, 225
- redefining a command, 55, 168, 192  
  with **\typein**, 212
- redefining an environment, 56, 192
- reducing a picture, 118
- \ref**, 67, 209  
  ~ used with, 68
- similar to **\cite**, 71
- undefined, 146
- value, 209  
  in **enumerate** environment, 184  
  in **list** environment, 186  
  in theorem-like environment, 193,  
    194  
  set by **\refstepcounter**, 195
- wrong number printed by, 146
- reference point, 103, 221  
  of picture object, 120
- reference undefined warning, 146, 147
- reference, circular, 243
- \reflectbox**, 130, 224
- reflection, 130
- \refstepcounter**, 195  
  **\ref** value set by, 209  
  resets counter, 194
- register, boolean, 196
- Reid, Brian, 7
- remark in citation, 210
- \renewcommand**, 55, 168, 192  
  error in, 141, 142
- \renewenvironment**, 56, 192  
  error in, 141
- repeated logical structure, 53
- repeated pattern in picture, 127–128
- report document class, 19, 176  
  appendix in, 175  
  default page style, 89  
  **thebibliography** environment in, 209  
  **titlepage** option the default, 88
- report.cls** file, 92
- required bibliography field, 160
- \resizebox**, 129–130, 224
- \resizebox\***, 224
- return** key, 12
- reusing a **picture** environment, 107
- \reversemarginpar**, 201
- \rfloor** (⟨), 47
- rgb** color model, 132, 225
- \rhd** (⟨), 42
- \rho** (⟨), 41
- right head for first page, 180
- right margin, prevailing, 203

\right, 47, 190  
 right, flush, *see* flush right  
 right-hand page, 179  
     starting on, 97  
 \Rightarrow ( $\Rightarrow$ ), 43  
 \rightarrow ( $\rightarrow$ ), 43  
 \rightharpoondown ( $\rightarrow$ ), 43  
 \rightharpoonup ( $\rightarrow$ ), 43  
 \rightleftharpoons ( $\rightleftharpoons$ ), 43  
 \rightmargin, 113, 185  
     in *trivlist* environment, 115  
 rigid length, 100  
 \rmfamily, 37, 226  
 robust command, 22, 167  
 roman page numbers, 89, 180  
 roman type family, 37, 226  
     in math mode, 51, 191  
     in *slides* document class, 81  
 \Roman, 98, 195  
 \roman, 98, 195  
 Romanian, 94  
 root file, 72  
 root, square, 40, 189  
     space around, 51  
 \rotatebox, 130, 224  
 \rotatebox\*, 224  
 rotation, 130  
 rounded corner, 124–125  
 rubber length, 100  
     in \hspace or \vspace, 102  
     in \lengthtest expression, 196  
     infinitely stretchable, 102, 215  
 rule (mathematical), 56  
 rule box, 103, 106  
 \rule, 106, 219  
     horizontal line drawn with, 106  
     vertical line drawn with, 106  
 running head, *see* head  
 running L<sup>A</sup>T<sub>E</sub>X  
     on part of document, 31, 73–74  
     unattended, 30  
  
 s positioning argument, 217  
 \S (§), 39, 173  
 Samarin, Alexander, 2  
 \samepage (L<sup>A</sup>T<sub>E</sub>X 2.09 command), 229  
 sample input, 2, 8  
  
 sample2e.tex file, 3, 8, 19, 28, 93  
 sans serif type family, 37, 226  
     in math mode, 51, 191  
 save size, 144  
 \savebox, 107, 218, 223  
     in *picture* environment, 120, 127  
     use of \width, \height, \depth, and  
         \totalheight in, 219  
 saving a box, 107–108, 127, 218  
 saving typing, 54  
 \sbox, 108, 218  
 \scalebox, 129–130, 224  
 scaling, 129  
 school field (in *bib* file), 163  
 scope of declaration, 27–28, 168  
     in argument of user-defined command  
         or environment, 55, 56, 192  
     in *array* environment, 45  
     in *tabbing* environment, 61, 203  
     nested too deeply, 144  
     within command argument, 193  
 screen output, *see* terminal  
 screen previewer, 6  
 script letter, 42  
 script math style, 52, 191  
 scriptscript math style, 52, 191  
 \scriptscriptstyle, 52, 191  
 \scriptsize, 115, 226  
 \scriptstyle, 52, 191  
 scroll, the, 135  
 \scshape, 37, 226  
 \searrow ( $\searrow$ ), 43  
 \sec (sec), 44  
 \secnumdepth counter, 174, 176  
 section counter, 97  
 section numbering, 176  
 section structure, 21  
 \section, 21, 174  
 sectional unit, 21  
     cross-reference to, 67  
     in different document classes, 21, 22  
     level number of, 176  
     numbering of, 174  
     numbering within, 57  
 sectioning command, 21–22, 174–176  
     \ in, 169  
     \*-form of, 174

- argument too long, 143  
blank line not allowed in, 171  
examples, 174  
fragile command in, 135  
in front and back matter of a book, 80  
`\label` in argument, 209  
moving argument of, 168  
optional argument of, 174  
page heading set by, 90, 174  
table of contents entry, 22, 174, 209  
“see” index entry, 152  
`\see`, 153  
sending a document, 77  
sentence, 13  
`series` field (in bib file), 163  
series of type, *see* type  
`\setboolean`, 196  
`\setcounter`, 98, 194  
    error in, 138, 144  
`\protect` not used in argument of, 168  
scope of, 98, 168  
`\setlength`, 101, 216  
`\setminus` (\), 42  
`\settabs` (TeX command), 232  
`\settime`, 83  
setting tab stops, 61  
`\settodepth`, 101, 216  
`\settoheight`, 101, 216  
`\settowidth`, 101, 216  
`\sevenbf` (TeX command), 233  
`\seveni` (TeX command), 233  
`\sevensy` (TeX command), 233  
`\sffamily`, 37, 226  
shape of type, 36  
`\sharp` (#), 43  
`\shortstack`, 123–124, 222  
    \\ in argument, 169  
showidx package, 75, 179  
`\Sigma` (\Sigma), 41  
`\sigma` (\sigma), 41  
sign, minus, .14  
`\signature`, 84  
`\sim` (\sim), 43  
`\simeq` (\simeq), 43  
simulating typed text, 63–64  
`\sin` (sin), 44  
single quote, 12, 13, 170  
single-column format, 181  
`\sinh` (sinh), 44  
size  
    buffer, 143  
    hash, 143  
    main memory, 143  
    of paper, 177  
    of type, *see* type  
    pool, 144  
    save, 144  
skinny, making things, 129  
skip (TeX term), 100  
slanted line, minimum size of, 123  
slanted type shape, 37, 226  
slash through symbol, 42  
slide environment, 81  
slides, 80–84, 228  
slides document class, 80–84, 176  
SLiTeX, 228  
slope of line in picture, 122  
`\sloppy`, 94, 214  
    causes underfull `\hbox` message, 147  
`\sloppypar` environment, 94, 214  
    causes underfull `\hbox` message, 147  
`\slshape`, 37, 226  
small caps type shape, 37, 226  
`\small`, 115, 226  
`small2e.tex` file, 2  
`\smallskip`, 216  
`\smallskipamount`, 216  
`\smile` (\smile), 43  
source list, 69, 70, 209  
source, bibliographic, 69  
`source2e.tex` file, 91  
space, 216–217  
    above displayed formula, 107, 188, 189  
    after punctuation, 14–15, 170, 171  
    after tabbing command, 61  
    around + and –, 48  
    around `\hspace` command, 101  
    around integral sign, 51  
    around square root, 51  
    at beginning or end of output line, 102  
    at top or bottom of page, 102  
    avoiding unwanted, 98, 109  
    below displayed formula, 107, 189  
    between paragraphs, 100, 172

- character, 13
  - at end of line, 19
  - ignored in math mode, 36
  - in command definition, 54
  - in formula, 18
  - in LR mode, 36
  - in `\put` argument, 129
  - in `\typeout` or `\typein` argument, 212
  - multiple, 13, 154
  - not allowed after `\verb` command, 64
  - not allowed in `\end{verbatim}`, 64
- ending command name with, 16
- ending line without adding, 33
- horizontal, *see* horizontal space
- ignored after command name, 16
- in array, 45
- in formula, 50–51
- intercolumn, *see* intercolumn space
- interrow, *see* interrow space
- interword, *see* interword space
- medium, 51, 191
- negative thin, 51, 191
- thick, 51, 191
- thin, 14, 33, 51, 170, 191
  - vertical, *see* vertical space
- `\space`, 212
- space-filling dots, 102
- space-filling horizontal line, 102
- spacing, 101–103
  - command, 170
  - double, 172
- `\spadesuit` (♠), 43
- special character, 12, 32, 171
  - in `\index` argument, 75, 153
  - in `\verb` environment, 64
  - used incorrectly, 31
- special font, 116
- special symbol, 38–39, 226
  - `\sqcap` (⊓), 42
  - `\sqcup` (⊔), 42
  - `\sqrt` (√), 40, 51, 189
  - `\sqsubset` (⊓), 43
  - `\sqsubseteq` (⊑), 43
  - `\sqsupset` (⊔), 43
  - `\sqsupseteq` (⊑), 43
- square bracket, 12
  - enclosing optional argument, 20, 33, 166
  - mistaken for optional argument, 142
- square root, 40, 189
  - space around, 51
- `\ss` (ß), 39
- stack, 123–124
- stacking symbols, 50
- `\stackrel`, 50, 191
  - making symbol with, 42
- `\star` (\*), 42
- `\stepcounter`, 195
  - resets counter, 194
- `\stop`, 30
  - last page produced by, 136
- stopping L<sup>A</sup>T<sub>E</sub>X, 30, 136
- storage bin, 107, 127
- `\stretch`, 215
- stretchable length, *see* rubber length
- structure
  - logical, 6, 88
  - of document, 170
  - section, 21
  - theorem-like, 56–57
- strut, 106
  - example of use, 110
  - in `array` and `tabular` environments, 169
- stupidity, in Unix, 32
- `sty` (package) file, 91, 166
  - © regarded as letter in, 91
  - missing, 137
- style
  - bibliography, 70–71
  - customizing, 91–93
  - math, *see* math
  - numbering, 98
  - page, *see* page
  - parameter, 166
  - type, *see* type
- subentry, index, 75
- `\subitem`, 75, 211
- subparagraph counter, 97
- `\subparagraph`, 21, 174
- subpicture, 128

- subscript, 18, 33, 40, 189  
  double, error, 140  
  math style for, 52, 191  
  of log-like function, 190  
  size of type in, 52, 116  
**subsection** counter, 97  
**\subsection**, 21, 174  
**\subset** ( $\subset$ ), 43  
**\subseteqq** ( $\subseteq$ ), 43  
substituted type style, 37  
subsubentry, index, 75  
**\subsubitem**, 75, 211  
subsubsection counter, 97  
**\subsubsection**, 21, 174  
**\succ** ( $\succ$ ), 43  
**\succeq** ( $\succeq$ ), 43  
**\sum** ( $\sum$ ), 44  
sundial, 94  
**\sup** ( $\sup$ ), 44  
  subscript of, 190  
superscript, 18, 33, 40, 189  
  double, error, 140  
  math style for, 52, 191  
  size of type in, 52, 116  
support, acknowledgment of, 181  
**\suppressfloats**, 199  
  optional argument of, 167  
**\supset** ( $\supset$ ), 43  
**\supseteqq** ( $\supseteq$ ), 43  
**\surd** ( $\sqrt$ ), 43  
**\swarrow** ( $\swarrow$ ), 43  
symbol  
  bar over, 49  
  bold, 51  
  footnote, 195  
  too many, 137  
making with **array** environment, 42  
making with **\stackrel**, 42  
math, 41–45  
  non-English, 38–39  
  in **bib** file, 158  
nonmath, 38–39  
not in typewriter style, 38  
not provided by **LATEX**, 116  
slash through, 42  
special, 38–39, 226  
stacking, 50  
variable-sized, 42, 52  
**\symbol**, 116, 226
- t** (top)  
  float specifier, 197  
oval-part argument, 124, 223  
positioning argument, 46, 105, 121, 218, 222  
**\t** (^ accent), 38  
tab overflow error, 139  
tab stop, 61, 201  
  too many, 139  
  undefined, 140  
tab, alignment, error, 141  
**\tabalign** (**TeX** command), 232  
tabbing command  
  Plain **TeX**, 232  
  space after, 61  
**tabbing** environment, 60–62, 201–203  
  \\ in, 169  
  as displayed paragraph, 183  
  example, 202  
  in **minipage** environment, 106, 218  
  in **parbox**, 105  
  large, 143  
  redefinition of commands in, 203  
  scope of declaration in, 61  
  used instead of **verbatim**, 64  
  versus **tabular**, 60  
**\tabbingsep**, 203  
**\tabcolsep**, 207  
table, 58–59, 197–200  
  centering, 112  
  in two-column format, 197  
  item, paragraph in, 104  
  made with **tabular** environment, 58  
  of contents, 66–67, 175–176  
  adding an entry, 175  
  adding commands to, 67, 176  
  depth, 176  
  entry made by sectioning  
    command, 22, 174  
  error in, 135  
  example, 174  
  generated from **toc** file, 209  
  placement of, *see* float  
  too tall, 145

**table** counter, 97  
**table** environment, 58–59, 197  
 in **parbox**, 138  
 misplaced, 138  
**parbox** made by, 104  
 space around, 169  
 too many, 139  
**table\*** environment, 197  
**\tableofcontents**, 66, 175–176  
 error when processing, 143  
**toc** file read by, 209  
 tables, list of, 67, 175–176  
**\tabs** (**TeX** command), 232  
**\tabsdone** (**TeX** command), 232  
**\tabset** (**TeX** command), 232  
**tabular** environment, 62–63, 204–207  
 \\ in, 169  
 box made by, 103  
 error in, 137, 138, 141, 145  
 example, 204  
 extra space around, 205  
 footnoting item of, 105  
 illegal character in argument, 137  
 in **parbox**, 218  
 interrow space in, 169  
 large, 143  
 making tables with, 58  
 strut in, 169  
 versus **array**, 46, 60  
 versus **tabbing**, 60  
**tabular\*** environment, 204–207  
**\tan** (**tan**), 44  
**\tanh** (**tanh**), 44  
**\tau** ( $\tau$ ), 41  
**techreport** bibliography entry type, 162  
 ten-point type, 19, 115  
**\teni** (**TeX** command), 233  
 term, invisible, 48  
 terminal, 28  
 defining command from, 76  
 input, 76–77, 212–213  
 message, spaces in, 212  
 output, 76–77, 212–213  
 written on **log** file, 208  
 printing command on, 212  
**TeX**, 5, 231–233  
 distinguished from **L<sup>A</sup>T<sub>E</sub>X**, 5  
 error message, 140–145  
 error versus **L<sup>A</sup>T<sub>E</sub>X** error, 29  
 logo, 5, 15, 33, 171  
 pronunciation of, 5  
 warning message, 147  
**\TeX**, 15, 33, 171  
**text**  
 editor, 12, 32  
 emphasized, 16, 171  
 file, 12, 144  
 invisible, 82, 97  
 math style, 52, 187  
 positioning of, 107, 121  
 typed, simulating, 63–64  
**text-generating command**, 15  
**\textbf**, 37, 226  
**\textcolor**, 131, 225  
 used for invisible text in slide, 82  
**\textfloatsep**, 200  
**\textfraction**, 200  
**\textheight**, 100, 182  
**\textit**, 37, 226  
**\textmd**, 37, 226  
**\textnormal**, 226  
**\textrm**, 37, 226  
**\textsc**, 37, 226  
**\textsf**, 37, 226  
**\textsl**, 37, 226  
**\textstyle**, 52, 191  
 used with **\stackrel**, 50  
**\texttt**, 37, 226  
**\textup**, 37, 226  
**\textwidth**, 100, 171, 182  
**\thanks**, 181  
 moving argument of, 168  
**\the...** numbering commands, 98, 195  
**thebibliography** environment, 71–72,  
 209  
 as list, 112  
 missing argument in, 138  
**theindex** environment, 75–76, 211  
 \item in, 75  
**theorem**, 56  
 named, 57  
**theorem-like** environment, 56–57,  
 193–194  
 as displayed paragraph, 183

- as list, 112
- counter for, 97, 193
- cross-reference to, 68
- theorem-like structure, 56–57
- `\thepage` redefined by `\pagenumbering`, 180
- `\Theta`, 41
- `\theta`, 41
- thick space, 51, 191
- `\thicklines`, 119, 223
- thickness, *see* width
- thin space, 14, 33, 51, 170, 191
- thin, making things, 129
- `\thinlines`, 118, 223
- `\thispagestyle`, 89–90, 179
  - scope of, 168
- tick marks, 184
- tilde (~), 12, 17, 32
- `\tilde` (~ math accent), 50
- time, printing on notes, 83
- `\times`, 42
- `\tiny`, 115, 226
- title, 20–21
  - acknowledgment of support in, 181
  - author's address in, 181
  - date in, 20
  - example, 183
  - in `bib` file, 158
  - page, 20–21, 88, 90, 177, 181–183
- title field (in `bib` file), 163
- `\title`, 20, 34, 181
  - \ in, 169
  - not given error, 138
- `\titlepage` document-class option, 88, 90, 177
- `\titlepage` environment, 90, 183
  - making blank page with, 180
- `toc` (table of contents) file, 66, 67, 175, 209
  - editing, 67
  - error in, 135
- `tocdepth` counter, 176
- `\today`, 15, 33, 171
  - in title page, 90
  - redefining in letters, 86
- `\top` (T), 43
- `\topfraction`, 199
- `\topins` (TeX command), 232
- `\topinsert` (TeX command), 232
- `\topmargin`, 182
- `topnumber` counter, 199
- `\topsep`, 113, 185
  - in `fleqn` option, 188, 189
- `\topskip`, 181
- `\totalheight`, 219
- `totalnumber` counter, 200
- Trahison des Images, La, 131
- transformation, geometric, 129
- `\triangle` (△), 43
- `\triangleleft` (⊣), 42
- `\triangleright` (⇒), 42
- trivlist environment, 112, 115, 186
- Truman, Harry, 24
- `\ttfamily`, 37, 226
- 12pt document-class option, 19
- twelve-point type, 19, 115
- two-column format, 20, 88, 180
  - `\cleardoublepage` in, 97
  - `\clearpage` in, 97
  - figures and tables in, 197
  - marginal notes in, 59
  - `\newpage` in, 97
  - `\pagebreak` in, 97
- two-sided printing, 19, 177
  - increasing page height in, 214
  - marginal notes in, 59
  - page heading in, 89
- `\twocolumn` document-class option, 20, 177
  - marginal notes in, 59
- `\twocolumn`, 88, 180
  - optional argument of, 167
  - optional argument too tall, 146
- `\twoside` document-class option, 19, 177
  - default in `book` class, 80
  - evens page bottoms, 88
  - marginal notes in, 59
  - page heading in, 90
- type
  - bibliography entry, 156, 160–162
  - eleven-point, 19, 115
  - family, 36
    - roman, 37, 51, 191, 226
    - sans serif, 37, 51, 191, 226

- typewriter, 37, 51, 191, 226  
 font, *see* font  
 series, 36  
     boldface, 37, 51, 191, 226  
     medium, 37, 226  
 shape, 36  
     italic, 37, 51, 191, 226  
     slanted, 37, 226  
     small caps, 37, 226  
     upright, 37, 226  
 size, 226  
     changing, 115–116  
     default, 115  
     in footnote, 116  
     of sub- and superscripts, 52, 116  
 style, 36–37, 225–226  
     calligraphic, 51, 191  
     in math mode, 51–52, 191  
     substituted, 37  
     unavailable, 37, 116  
 ten-point, 19, 115  
 twelve-point, 19, 115  
 type field (in bib file), 164  
 typed text, simulating, 63–64  
 \typein, 76, 212  
     moving argument of, 77, 168  
     of \includeonly, 74  
 \typeout, 76, 212  
     moving argument of, 77, 168  
 typesetter, 6  
 typewriter type family, 37, 226  
     in math mode, 51, 191  
     no accents or symbols in, 38  
 typewriter, simulating, 63–64  
 typing, saving, 54  
 typographic  
     design, 91  
     designer, 5  
     error, 91  
  
 \u (‘ accent), 38  
 unary + and –, 48  
 unavailable type style, 37, 116  
 unbalanced braces, error caused by, 136  
 \unboldmath, 191  
 undefined  
     citation, 145, 147  
  
 color, 140  
 color model, 140  
 control sequence error, 144  
 environment error, 137  
 \pageref, 146  
 \ref, 146  
 reference, 146, 147  
 tab position, 140  
 \underbrace, 49  
 underfull \hbox message, 95, 147  
     caused by \\ and \newline, 213  
     caused by \linebreak, 147, 213  
     caused by \sloppy, 147, 214  
     caused by \sloppypar, 147  
 underfull \vbox message, 147  
     caused by \pagebreak, 214  
 \underline, 49, 190  
 underlining, 49, 190  
 unit  
     length, 118  
     of length, 215  
     of measure, illegal, 141  
     paragraph, 94  
     sectional, *see* sectional unit  
 \unitlength, 118, 219  
     for subpictures, 129  
 Unix, e-mail in, 32  
 Unix, stupidity in, 32  
 unknown graphics extension, 140  
 unknown option, 140  
 \unlhd (⊑), 42  
 unmatched  
     \$ (dollar sign), 141  
     }, 141  
     \begin, 136  
     brace, 31  
     \end, 136  
     math mode delimiter, 141  
     \poptabs, 138  
     \pushtabs, 138  
 unpublished bibliography entry type, 162  
 \unrhd (⊒), 42  
 unsrt bibliography style, 70  
 \Uparrow (↑), 43, 47  
 \uparrow (↑), 43, 47  
 \Updownarrow (↕), 43, 47

- \updownarrow** (↑), 43, 47  
**\uplus** (⊕), 42  
 uppercase letter, 12  
 upright type shape, 37, 226  
**\upshape**, 37, 226  
**\Upsilon** (Υ), 41  
**\upsilon** (υ), 41  
 use doesn't match definition error, 145  
**\usebox**, 107, 218  
**\usecounter**, 114, 186  
**\usepackage**, 20, 34, 170, 178–179  
     misplaced, 136  
     missing, 144  
     option obtained from **\documentclass**  
         command, 178  
     redundant, 138  
 user-defined command or environment,  
     scope in argument of, 55, 56, 192
- \v** (ˇ accent), 38  
 value of counter, 194  
**\value**, 194  
     in **ifthen** package expression, 195  
 van Leunen, Mary-Claire, 8  
**\varepsilon** (ε), 41  
 variable-sized math symbol, 42, 52  
 variant Greek letters, 41  
**\varphi** (φ), 41  
**\varpi** (ϖ), 41  
**\varrho** (ϱ), 41  
**\varsigma** (ς), 41  
**\vartheta** (ϑ), 41  
**\vbox**, overfull, 147  
**\vbox**, underfull, 147, 214  
**\vdash** (⊣), 43  
**\vdots** (⋮), 41, 189  
**\vec** (ˇmath accent), 50  
**\vector**, 123, 222  
     error in, 136  
**\vee** (∨), 42  
**\verb**, 64, 187  
     in argument of a command, 140  
     text ended by end of line, 140  
**\verb\***, 64, 187  
**\verbatim** environment, 63–64, 186  
**\verbatim\*** environment, 64, 186
- \verse** environment, 25–26, 34, 184  
     \\ in, 169  
     as list, 112  
 version, of **LATEX**, 2, 228–229  
**\Vert** (||), 43, 47  
**\vert** (|), 47  
 vertical  
     alignment, 46, 60–63, 201–207  
     ellipsis, 40, 189  
     line  
         drawn with **\rule**, 106  
         in **tabular** environment, 62  
     mode, 36  
     positioning  
         of **array** environment, 46  
         of array item, 46  
         of text, 107, 121, 219  
 space, 102  
     above displayed formula, 188, 189,  
         218  
     adding, 106, 216  
     at top or bottom of page, 102  
     below displayed formula, 183, 189  
         in math formula, 106  
**\vfill**, 102  
**\vfootnote** (**TEx** command), 232  
 visual design, 7, 88  
     of marginal notes, 60  
 visual formatting, 64  
     for slides, 81  
     of formula, 49, 52  
     with boxes, 108–111  
 visual property, 37  
**\vline**, 206  
**\volume** field (in **bib** file), 164  
**\vspace**, 102, 216  
     in figure, 58  
     removing space with, 184  
     rubber length in, 102  
     space around, 169  
     using strut instead, 106  
**\vspace\***, 102, 216
- warning message, 30  
     **LATEX**, 145–147  
     **TEx**, 147  
**\wedge** ( ∧ ), 42

\whiledo, 197  
white, 132, 225  
wide math accent, 49  
\widehat (math accent), 49, 190  
\widetilde (math accent), 49, 190  
width  
    box with specified, 104  
    of a box, 103  
    of line, 91, 171  
        for \fbox and \framebox, 219  
        in array or tabular environment, 207  
    in picture, 118, 223  
    of paragraph indentation, 99  
    of text on page, 100, 171  
\width, 219  
Wiles, Andrew, 57  
window, 28  
word, 13  
    index, 150  
\wp ( $\wp$ ), 43  
\wr ( $\wr$ ), 42  
writing, 8  
wrong number printed by \cite,  
    \pageref, and \ref, 146  
WYSIWYG, 7  
  
\Xi ( $\Xi$ ), 41  
\xi ( $\xi$ ), 41  
  
year field (in bib file), 164  
yellow, 132, 225  
  
zero (0), 12  
zero length, 99  
zero-length arrow, 123  
zero-width box, 121  
\zeta ( $\zeta$ ), 41

## Line Breaking

```
\linebreak force a line break
\\[len] start new line and leave len vertical space
\-- permit hyphenation
\begin{sloppypar} ... \end{sloppypar}
 allow loose lines in paragraphs
\sloppy allow loose lines
```

## Page Breaking

```
\pagebreak force a page break
\enlargethispage*{ht} squeezes extra ht of
 text on current page.
\newpage start a new page
\clearpage print all figures and tables and start
 a new page
```

## Boxes

```
\mbox{...}
\makebox[wd][pos]{...}
 make box of width wd; pos puts text at
 left (l), right (r), or center (default)
\fbox[text]
\framebox[wd][pos]{text}
 same as \mbox or \makebox but draws frame
 around box
\newsavebox{cmd} defines cmd to be a bin for
 saving boxes
\sbox{cmd}{text}
\savebox{cmd}[wd][pos]{text}
 same as \mbox or \makebox but saves box in
 bin cmd
\usebox{cmd} print box saved in bin cmd
\begin{minipage}[pos][wd]{...} \end{...}
 make parbox of width wd, aligned by pos at
 top (t), bottom (b), or center (default) line
\parbox[pos][wd]{...} same as minipage for
 small text, no displayed environments
```

## Space

```
\hspace{len} make len horizontal space; *-form
 works even at beginning of line
\hfill make infinitely stretchable horizontal
 space
\vspace{len} leave len vertical space; *-form
 works even at beginning of para
```

## Length

```
units cm em ex in pc pt mm
\newlength{cmd} define cmd to be a length
\setlength{cmd}{len} set length cmd to len
\addtolength{cmd}{len} add len to length cmd
```

## Pictures

```
\begin{picture}{x,y}(x',y') ... \end{...}
 x × y picture [lower-left corner at (x',y')]
\put(x,y){...} put object at point (x,y)
\multiput(x,y)(Δx,Δy){n}{...}
 make n copies of object with first at (x,y) and
 others offset by (Δx,Δy)
\makebox(x,y)[pos]{...} make x × y box; pos
 puts object at top (t), bottom (b), left
 (l), right (r), and/or centered (default);
 \framebox and \savebox have analogs
\dashbox(d)(x,y)[pos]{...} like \makebox but
 puts dashed lines of length d around box
\line(h,v){l} line of slope v/h and horizontal
 extent l (length l if h = 0), $0 \leq h, v \leq 6$
\vector(h,v){l} same as \line but draws ar-
 rowhead; $0 \leq h, v \leq 4$
\shortstack[pos]{...}
 like \begin{tabular}[pos]{...}
\circle{d} draw circle of diameter d; *-form
 draws solid disk
\oval(x,y)[part] draw x × y [partial] oval
\frame{...} draw frame around object
\line thickness \thinlines or \thicklines
```

## graphics and color Packages

```
\scalebox{fac}{...} scale by factor of fac
\resizebox{wd}{ht}{...} scale to wd × ht
\rotatebox{ang}{...} rotate by ang degrees
\includegraphics{file} insert graphics from file
\definecolor{clr}{mdl}{val} define color clr
 using color model mdl
\color{clr} set current color to clr
\textcolor{clr}{...} typeset in color clr
\colorbox{clr}{...} typeset on colored box
\pagecolor{clr} set background color of page
```

## Figures and Tables

```
\begin{figure} ... \end{figure}
 make floating figure
\begin{table} ... \end{table}
 make floating table
\caption{...} make figure or table caption
```

## tabbing Environment

```
Rows separated by \\; columns determined by:
\= set tab stop
\> go to next tab stop
\kill throw away line
```

## array and tabular Environments

```
\begin{array}[pos][cols]{...} \end{array}
\begin{tabular}[pos][cols]{...} \end{tabular}
 use array for formulas, tabular for text; items
 separated by & and rows by \\; pos aligns with
 top (t), bottom (b), or center (default); cols
 entries format columns:
 l left-justified column
 r right-justified column
 c centered column
 | vertical rule
 \text{...} text or space between columns
 *{n}{...} equivalent to n copies of ...
\multicolumn{n}{col}{...} span next n col-
 umns with col format
\hline draw horizontal line between rows
\cline{i-j} horizontal line across columns i-j
```

## Definitions

```
\newcommand{cmd}[n][opt]{...} define com-
 mand cmd [with n arguments] [first optional]
\newenvironment{nam}[n]{beg}{end}
 define environment nam [with n arguments]
\newtheorem{nam}[cap] define a theorem-like
 environment nam captioned by cap
```

## Numbering

```
\setcounter{ctr}{n} set counter ctr to n
\addtocounter{ctr}{n} add n to counter ctr
```

## Sentences and Paragraphs

```
quotes single '...' double “...”
dashes intra-word - number range: --
punctuation: ---
spacing small \, inter-word _ unbreakable - sentence-ending period \@.
special characters $ \$ & \& % \%
\# { \{ } \} - _
emphasis \emph{...}
unbreakable text \mbox{...}
footnotes \footnote{...}
date \today
```

## Type Style

|              |      |              |      |
|--------------|------|--------------|------|
| \textrm{...} | Rom  | \textsc{...} | CAPS |
| \textit{...} | Ital | \texttt{...} | Type |
| \textbf{...} | Bold | \textsf{...} | SSrf |
| \textsl{...} | Slan |              |      |

\boldmath use bold math symbols  
in math mode

|              |      |               |      |
|--------------|------|---------------|------|
| \mathrm{...} | Rom  | \mathit{...}  | Type |
| \mathit{...} | Ital | \mathsf{...}  | SSrf |
| \mathbf{...} | Bold | \mathcal{...} | CAL  |

## Type Size

|               |             |        |       |
|---------------|-------------|--------|-------|
| \tiny         | \small      | \large | \huge |
| \scriptsize   | \normalsize | \Large | \Huge |
| \footnotesize |             | \LARGE |       |

## Accents and Symbols

|         |          |         |         |
|---------|----------|---------|---------|
| ö \'{o} | ö \\"{o} | ö \v{o} | ö \c{o} |
| ó \'{o} | ó \=o}   | ó \H{o} | ó \d{o} |
| ô \^o}  | ô \.o}   | ô \t{o} | ô \b{o} |
| ó \"o}  | ó \u{o}  |         |         |

|         |      |              |  |
|---------|------|--------------|--|
| † \dag  | § \S | © \copyright |  |
| ‡ \ddag | ¶ \P | £ \pounds    |  |

## Sectioning and Table of Contents

|          |                |               |
|----------|----------------|---------------|
| \part    | \section       | \paragraph    |
| \chapter | \subsection    | \subparagraph |
|          | \subsubsection |               |

\appendix start appendix  
\tableofcontents make table of contents

## Mathematical Formulas

\\$...\$ or \(...\)\_ in-text formula  
\[...]\ displayed formula  
\begin{equation} ... \end{equation} numbered equation  
\begin{eqnarray} ... \end{eqnarray} numbered equations, like 3-column array environment; \nonumber omits one equation number, eqnarray\* omits all  
\{...} subscript  
\{...} superscript  
\prime ()  
\frac{n}{d} print fraction  $\frac{n}{d}$   
\sqrt[n]{arg} print  $\sqrt[n]{arg}$   
ellipsis \ldots \cdots \vdots  
symbols see Tables 3.4–3.8 (pp. 42–44)  
Greek letters \alpha \alpha ... \Omega \Omega  
delimiters \left or \right followed by delimiter from Table 3.10 (p. 47)  
\overline{exp} print  $\overline{exp}$   
space thin \, medium \: thick \; negative thin \!

## Displayed Paragraphs

\begin{quote} ... \end{quote} short displayed quotation  
\begin{quotation} ... \end{quotation} long displayed quotation  
\begin{center} ... \end{center} centered lines, separated by \\  
\begin{verse} ... \end{verse} \\ between lines, blank line between stanzas  
\begin{verbatim} ... \end{verbatim} in typewriter font exactly as formatted

## Lists

Begin each item with \item or \item[*label*]  
\begin{itemize} ... \end{itemize} “ticked” items  
\begin{enumerate} ... \end{enumerate} numbered items  
\begin{description} ... \end{description} labeled items

## Document Class, Packages, Styles

```
\documentclass[options]{class}
style article report book
slides letter (for letters)
options 11pt titlepage twoside leqno
12pt twocolumn a4paper fleqn
\usepackage[options]{pkg}
pkg amstex color latexsym
babel graphics makeidx
\pagestyle{style} style of head and foot:
plain empty headings myheadings
\pagenumbering{style} style of page numbers:
arabic roman alph Roman Alph
```

## Title Page and Abstract

```
\maketitle make title with information declared
by \title, \author, and [optional] \date
\begin{titlepage} ... \end{titlepage}
do-it-yourself title page
\begin{abstract} ... \end{abstract}
make abstract
```

## Cross-Reference

```
\label{key} assign current counter value to key
\ref{key} print value assigned to key
```

## Bibliography and Citation

```
\bibliography{...} make bibliography and tell
BESTEX names of bib files
\begin{thebibliography}{lbl} ... \end{...}
make bibliography; lbl is widest entry label
\bibitem[lbl]{key} begin bibliography entry for
citation key [with lbl as label]
\cite[note]{keys} cite reference(s) keys [with
added note]
```

## Splitting the Input

```
\input{file} read specified file
\include{file} read specified file unless excluded
by \includeonly
\includeonly{files} exclude any file not in files
\begin{filecontents}{file} contents \end{...}
write contents on specified file
```

**LATEX** is a software system for typesetting documents. Because it is especially good for technical documents and is available for almost any computer system, **LATEX** has become a *lingua franca* of the scientific world. Researchers, educators, and students in universities, as well as scientists in industry, use **LATEX** to produce professionally formatted papers, proposals, and books. They also use **LATEX** input to communicate information electronically to their colleagues around the world.

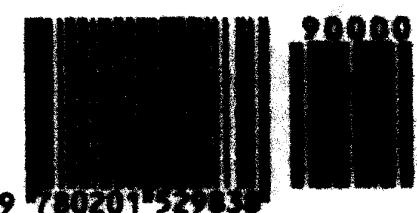
With the release of **LATEX 2<sub>E</sub>**, the new standard version, **LATEX** has become even more powerful. Among its new features are an improved method for handling different styles of type, and commands for including graphics and producing colors. **LATEX 2<sub>E</sub>** makes available to *all* **LATEX** users valuable enhancements to the software that have been developed over the years by users in many different places to satisfy a variety of needs.

This book, written by the original architect and implementer of **LATEX**, is both the user's guide and the reference manual for the software. It has been updated to reflect the changes in the new release. The book begins with instructions for formatting simpler text, and progressively describes commands and techniques for handling larger and more complicated documents. A separate chapter explains how to deal with errors. An added appendix describes what is new and different in **LATEX 2<sub>E</sub>**. Other additions to the second edition include:

- Descriptions of new commands for inserting pictures prepared with other programs and for producing colored output;
- New sections on how to make books and slides;
- Instructions for making an index with the *MakeIndex* program, and an updated guide to preparing a bibliography with the *BIBLEX* program;
- A section on how to send your **LATEX** documents electronically.

Users new to **LATEX** will find here a book that has earned worldwide praise as a model for clear, concise, and practical documentation. Experienced users will want to update their **LATEX** library. Although most standard **LATEX** input files will work with **LATEX 2<sub>E</sub>**, a few of the new features, a few **LATEX 2<sub>E</sub>** conventions must first be learned. For those who want a more advanced guide to **LATEX 2<sub>E</sub>** and to more than 150 packages that can now be used with it, see *More **LATEX** Techniques* by Bach, and Samarin (also published by Addison-Wesley).

Leslie Lamport is a computer scientist well known for his contributions to concurrent computing, as well as for creating the **LATEX** typesetting system in 1985. He now works at the Systems Research Center of Digital Equipment Corporation. He received a Ph.D. in mathematics from Brandeis University.



ADDISON-WESLEY PUBLISHING COMPANY

ISBN 0-201-52963-1

## On interprocess communication

### Part I: Basic formalism\*

Leslie Lamport

Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA



Dr. Lamport is a member of Digital Equipment Corporation's Systems Research Center. In previous incarnations, he was with SRI International and Massachusetts Computer Associates. The central topic of his research has been concurrency, and he can write TEX macros and chew gum at the same time.

**Abstract.** A formalism for specifying and reasoning about concurrent systems is described. Unlike more conventional formalisms, it is not based upon atomic actions. A definition of what it means for one system to implement a higher-level system is given and justified. In Part II, the formalism is used to specify several classes of interprocess communication mechanisms and to prove the correctness of algorithms for implementing them.

**Key words:** Concurrent reading and writing – Nonatomic operations – Shared data

### Introduction

This is the first part of a two-part paper addressing what I believe to be fundamental ques-

tions in the theory of interprocess communication. It develops a formal definition of what it means to implement one system with a lower-level one and provides a method for reasoning about concurrent systems. The definitions and axioms introduced here are applied in Part II [5] to algorithms that implement certain interprocess communication mechanisms.

To motivate the formalism, let us consider the question of atomicity. Most treatments of concurrent processing assume the existence of atomic operations – an atomic operation being one whose execution is performed as an indivisible action. The term *operation* is used to mean a class of actions such as depositing money in a bank account, and the term *operation execution* to mean one specific instance of executing such an action – for example, depositing \$100 in account number 14335 at 10:35 a.m. on December 14, 1987. Atomic operations must be implemented in terms of lower-level operations. A high-level language may provide a *P* operation to a semaphore as an atomic operation, but this operation must be implemented in terms of lower-level machine-language instructions. Viewed at the machine-language level, the semaphore operation is not atomic. Moreover, the machine-language operations must ultimately be implemented with circuits in which operations are manifestly non-atomic – the possibility of harmful “race conditions” shows that the setting and the testing of a flip-flop are not atomic actions.

Part II considers the problem of implementing atomic operations to a shared register with more primitive, nonatomic operations. Here, a more familiar example of implementing atomicity is used: concurrency control in a database. In a database system, higher-level transactions,

\* Much of this research was performed while the author was a member of the Computer Science Laboratory at SRI International, where it was sponsored by the Office of Naval Research Project under contract number N00014-84-C-0621 and the Rome Air Development Command Project under contract number F30602-85-C-0024

which may read and modify many individual data items, are implemented with lower-level reads and writes of single items. These lower-level read and write operations are assumed to be atomic, and the problem is to make the higher-level transactions atomic. It is customary to say that a semaphore operation is atomic while a database transaction *appears to be* atomic, but this verbal distinction has no fundamental significance.

In database systems, atomicity of transactions is achieved by implementing a *serializable* execution order. The lower-level accesses performed by the different transactions are scheduled so that the net effect is the same as if the transactions had been executed in some serial order – first executing all the lower-level accesses comprising one transaction, then executing all the accesses of the next transaction, and so on. The transactions should not actually be scheduled in such a serial fashion, since this would be inefficient; it is necessary only that the effect be the same as if that were done.<sup>1</sup>

In the literature on concurrency control in databases, serializability is usually the only correctness condition that is stated [1]. However, serializability by itself does not ensure correctness. Consider a database system in which each transaction either reads from or writes to the database, but does not do both. Moreover, assume that the system has a finite lifetime, at the end of which it is to be scrapped. Serializability is achieved by an implementation in which reads always return the initial value of the database entries and writes are simply not executed. This yields the same results as a serial execution in which one first performs all the read transactions and then all the writes. While such an implementation satisfies the requirement of serializability, no one would consider it to be correct.

This example illustrates the need for a careful examination of what it means for one system to implement another. It is reconsidered in Sect. 2, where the additional correctness condition needed to rule out this absurd implementation is stated.

<sup>1</sup> In the context of databases, atomicity often denotes the additional property that a failure cannot leave the database in a state reflecting a partially completed transaction. In this paper, the possibility of failure is ignored, so no distinction between atomicity and serializability is made.

## 1 System executions

Almost all models of concurrent processes have indivisible atomic actions as primitive elements. For example, models in which a process is represented by a sequence or “trace” [10, 12, 13] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [9, 11] assume that the firing of an individual transition is atomic. These models are not appropriate for studying such fundamental questions as what it means to implement an atomic operation, in which the nonatomicity of operations must be directly addressed.

More conventional formalisms are therefore eschewed in favor of one introduced in [4] and refined in [3], in which the primitive elements are *operation executions* that are not assumed to be atomic. This formalism is described below; the reader is referred to [4] and [3] for more details.

A *system execution* consists of a set of *operation executions*, together with certain temporal precedence relations on these operation executions. Recall that an *operation execution* represents a single execution of some operation. When all operations are assumed to be atomic, an *operation execution*  $A$  can influence another *operation execution*  $B$  only if  $A$  precedes  $B$  – meaning that all actions of  $A$  are completed before any action of  $B$  is begun. In this case, one needs only a single temporal relation  $\rightarrow$ , read “precedes”, to describe the temporal ordering among *operation executions*. While temporal precedence is usually considered to be a total ordering of atomic operations, in distributed systems it is best thought of as an irreflexive partial ordering (see [6]).

Nonatomicity introduces the possibility that an *operation execution*  $A$  can influence an *operation execution*  $B$  without preceding it; it is necessary only that some action of  $A$  precede some action of  $B$ . Hence, in addition to the precedence relation  $\rightarrow$ , one needs an additional relation  $\rightarrow\rightarrow$ , read “can affect”, where  $A \rightarrow\rightarrow B$  means that some action of  $A$  precedes some action of  $B$ .

*Definition 1.* A *system execution* is a triple  $\langle \mathcal{S}, \rightarrow, \rightarrow\rightarrow \rangle$ , where  $\mathcal{S}$  is a finite or countably infinite set whose elements are called *operation executions*, and  $\rightarrow$  and  $\rightarrow\rightarrow$  are precedence relations on  $\mathcal{S}$  satisfying axioms A1-A5 below.

To assist in understanding the axioms for the  $\rightarrow$  and  $\dashrightarrow$  relations, it is helpful to have a semantic model for the formalism. The model to be used is one in which an operation execution is represented by a set of primitive actions or events, where  $A \rightarrow B$  means that all the events of  $A$  precede all the events of  $B$ , and  $A \dashrightarrow B$  means that some event of  $A$  precedes some event of  $B$ . Letting  $\mathbf{E}$  denote the set of all events, and  $\rightarrow$  the temporal precedence relation among events, we get the following formal definition.

**Definition 2.** A *model* of a system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  consists of a triple  $\mathbf{E}, \rightarrow, \mu$ , where  $\mathbf{E}$  is a set,  $\rightarrow$  is an irreflexive partial ordering on  $\mathbf{E}$ , and  $\mu$  is a mapping that assigns to each operation execution  $A$  of  $\mathcal{S}$  a nonempty subset  $\mu(A)$  of  $\mathbf{E}$ , such that for every pair of operation executions  $A$  and  $B$  of  $\mathcal{S}$ :

$$A \rightarrow B \equiv \forall a \in \mu(A): \forall b \in \mu(B): a \rightarrow b$$

$$A \dashrightarrow B \equiv \exists a \in \mu(A): \exists b \in \mu(B): a \rightarrow b \text{ or } a = b. \quad (1)$$

Note that the same symbol  $\rightarrow$  denotes the “precedes” relation both between operation executions in  $\mathcal{S}$  and between events in  $\mathbf{E}$ .

Other than the existence of the temporal partial-ordering relation  $\rightarrow$ , no assumption is made about the structure of the set of events  $\mathbf{E}$ . In particular, operation executions may be modeled as infinite sets of events. An important class of models is obtained by letting  $\mathbf{E}$  be the set of events in four-dimensional spacetime, with  $\rightarrow$  the “happens before” relation of special relativity, where  $a \rightarrow b$  means that it is temporally possible for event  $a$  to causally affect event  $b$ .

Another simple and useful class of models is obtained by letting  $\mathbf{E}$  be the real number line and representing each operation execution  $A$  as a closed interval.

**Definition 3.** A *global-time model* of a system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  is one in which  $\mathbf{E}$  is the set of real numbers,  $\rightarrow$  is the ordinary  $<$  relation, and each set  $\mu(A)$  is of the form  $[s_A, f_A]$  with  $s_A < f_A$ .

Think of  $s_A$  and  $f_A$  as the starting and finishing times of  $A$ . In a global-time model,  $A \rightarrow B$  means that  $A$  finishes before  $B$  starts, and  $A \dashrightarrow B$  means that  $A$  starts before (or at the same time as)  $B$  finishes. These relations are illustrated by Fig. 1, where operation executions

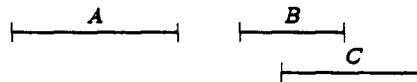


Fig. 1. Three operation executions in a global-time model

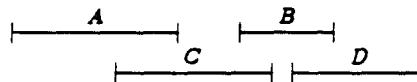


Fig. 2. An illustration of Axiom A4

$A$ ,  $B$ , and  $C$ , represented by the three indicated intervals, satisfy:  $A \rightarrow B$ ,  $A \rightarrow C$ ,  $B \dashrightarrow C$ , and  $C \dashrightarrow B$ . (In this and similar figures, the number line runs from left to right, and overlapping intervals are drawn one above the other.)

To complete Definition 1, the axioms for the precedence relations  $\rightarrow$  and  $\dashrightarrow$  of a system execution must be given. They are the following, where  $A$ ,  $B$ ,  $C$ , and  $D$  denote arbitrary operation executions in  $\mathcal{S}$ . Axiom A4 is illustrated (in a global-time model) by Fig. 2; the reader is urged to draw similar pictures to help understand the other axioms.

- A1. The relation  $\rightarrow$  is an irreflexive partial ordering.
- A2. If  $A \rightarrow B$  then  $A \dashrightarrow B$  and  $B \not\rightarrow A$ .
- A3. If  $A \rightarrow B \dashrightarrow C$  or  $A \dashrightarrow B \rightarrow C$  then  $A \dashrightarrow C$ .
- A4. If  $A \rightarrow B \dashrightarrow C \rightarrow D$  then  $A \rightarrow D$ .
- A5. For any  $A$ , the set of all  $B$  such that  $A \not\rightarrow B$  is finite.

(These axioms differ from the ones in [3] because only terminating operation executions are considered here.)

Axioms A1-A4 follow from (1), so they do not constrain the choice of a model. Axiom A5 does not follow from (1); it restricts the class of allowed models. Intuitively, A5 asserts that a system execution begins at some point in time, rather than extending into the infinite past. When  $\mathbf{E}$  is the set of events in space-time, A5 holds for any model in which: (i) each operation occupies a finite region of space-time, (ii) any finite region of space-time contains only a finite number of operation executions, and (iii) the system is not expanding faster than the speed of light.<sup>2</sup>

Most readers will find it easiest to think about

<sup>2</sup> A system expanding faster than the speed of light could have an infinite number of operation executions none of which are preceded by any operation

system executions in terms of a global-time model, and to interpret the relations  $\rightarrow$  and  $\dashrightarrow$  as indicated by the example in Fig. 1. Such a mental model is adequate for most purposes. However, the reader should be aware that in a system execution having a global-time model, for any distinct operation executions  $A$  and  $B$ , either  $A \rightarrow B$  or  $B \dashrightarrow A$ . (In fact, this is a necessary and sufficient condition for a system execution to have a global-time model [8].) However, in a system execution without a global-time model, it is possible for neither  $A \rightarrow B$  nor  $B \dashrightarrow A$  to hold. As a trivial counterexample, let  $\mathcal{S}$  consist of two elements and let the relations  $\rightarrow$  and  $\dashrightarrow$  be empty.

While a global-time model is a valuable aid to acquiring an intuitive understanding of a system, it is better to use more abstract reasoning when proving properties of systems. The relations  $\rightarrow$  and  $\dashrightarrow$  capture the essential temporal properties of a system execution, and A1-A5 provide the necessary tools for reasoning about these relations. It has been my experience that proofs based upon these axioms are simpler and more instructive than ones that involve modeling operation executions as sets of events.

## 2 Hierarchical views

A system can be viewed at different levels of detail, with different operation executions at each level. Viewed at the customer's level, a banking system has operation executions such as *deposit \$1000*. Viewed at the programmer's level, this same system executes operations such as *dep\_amt[cust]:=1000*. The fundamental problem of system building is to implement one system (like a banking system) as a higher-level view of another system (like a Pascal program).

A higher-level operation consists of a set of lower-level operations - the set of operations that implement it. Let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be a system execution and let  $\mathcal{H}$  be a set whose elements, called *higher-level operation executions*, are sets of operation executions from  $\mathcal{S}$ . A model for  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  represents each operation execution in  $\mathcal{S}$  by a set of events. This gives a representation of each higher-level operation execution  $H$  in  $\mathcal{H}$  as a set of events - namely, the set of all events contained in the representation of the lower-level operation executions that comprise  $H$ . This in turn defines precedence relations  $\xrightarrow{*}$  and  $\dashxrightarrow{*}$ , where  $G \xrightarrow{*} H$  means that all events in

(the representation of)  $G$  precede all events in  $H$ , and  $G \dashxrightarrow{*} H$  means that some event in  $G$  precedes some event in  $H$ , for  $G$  and  $H$  in  $\mathcal{H}$ .

To express all this formally, let  $\mathbf{E}, \rightarrow, \mu$  be a model for  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ , define the mapping  $\mu^*$  on  $\mathcal{H}$  by

$$\mu^*(H) = \bigcup \{\mu(A) : A \in H\}$$

and define the precedence relations  $\xrightarrow{*}$  and  $\dashxrightarrow{*}$  on  $\mathcal{H}$  by

$$G \xrightarrow{*} H \equiv \forall g \in \mu^*(G) : \forall h \in \mu^*(H) : g \rightarrow h$$

$$G \dashxrightarrow{*} H \equiv \exists g \in \mu^*(G) : \exists h \in \mu^*(H) : g \rightarrow h \text{ or } g = h.$$

Using (1), it is easy to show that these precedence relations are the same ones obtained by the following definitions:

$$G \xrightarrow{*} H \equiv \forall A \in G : \forall B \in H : A \rightarrow B$$

$$G \dashxrightarrow{*} H \equiv \exists A \in G : \exists B \in H : A \dashrightarrow B \text{ or } A = B. \quad (2)$$

Observe that  $\xrightarrow{*}$  and  $\dashxrightarrow{*}$  are expressed directly in terms of the  $\rightarrow$  and  $\dashrightarrow$  relations on  $\mathcal{S}$ , without reference to any model. We take (2) to be the definition of the relations  $\xrightarrow{*}$  and  $\dashxrightarrow{*}$ .

For the triple  $\langle \mathcal{H}, \xrightarrow{*}, \dashxrightarrow{*} \rangle$  to be a system execution, the relations  $\xrightarrow{*}$  and  $\dashxrightarrow{*}$  must satisfy axioms A1-A5. If each element of  $\mathcal{H}$  is assumed to be a nonempty set of operation executions, then Axioms A1-A4 follow from (2) and the corresponding axioms for  $\rightarrow$  and  $\dashrightarrow$ . For A5 to hold, it is sufficient that each element of  $\mathcal{H}$  consist of a finite number of elements of  $\mathcal{S}$ , and that each element of  $\mathcal{S}$  belong to a finite number of elements of  $\mathcal{H}$ . Adding the natural requirement that every lower-level operation execution be part of some higher-level one, this leads to the following definition.

**Definition 4.** A *higher-level view* of a system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  consists of a set  $\mathcal{H}$  such that:

- H1. Each element of  $\mathcal{H}$  is a finite, nonempty set of elements of  $\mathcal{S}$ .
- H2. Each element of  $\mathcal{S}$  belongs to a finite, nonzero number of elements of  $\mathcal{H}$ .

In most cases of interest,  $\mathcal{H}$  is a partition of  $\mathcal{S}$ , so each element of  $\mathcal{S}$  belongs to exactly one element of  $\mathcal{H}$ . However, Definition 4 allows the more general case in which a single lower-level operation execution is viewed as part of the implementation of more than one higher-level one.

Let us now consider what it should mean

for one system to implement another. If the system execution  $\langle S, \rightarrow, \dashrightarrow \rangle$  is an implementation of a system execution  $\langle H, \xrightarrow{H}, \xleftarrow{H} \rangle$ , then we expect  $H$  to be a higher-level view of  $S$  - that is, each operation in  $H$  should consist of a set of operation executions of  $S$  satisfying  $H_1$  and  $H_2$ . This describes the elements of  $H$ , but not the precedence relations  $\xrightarrow{H}$  and  $\xleftarrow{H}$ . What should those relations be?

If we consider the operation executions in  $S$  to be the "real" ones, and the elements of  $H$  to be fictitious groupings of the real operation executions into abstract, higher-level ones, then the induced precedence relations  $\xrightarrow{H}$  and  $\xleftarrow{H}$  represent the "real" temporal relations on  $H$ . These induced relations make the higher-level view  $H$  a system execution, so they are an obvious choice for the relations  $\xrightarrow{H}$  and  $\xleftarrow{H}$ . However, as we shall see, they may not be the proper choice.

Let us return to the problem of implementing atomic database operations. Atomicity requires that, when viewed at the level at which the operation executions are the transactions, the transactions appear to be executed sequentially. In terms of our formalism, the correctness condition is that, in any system execution  $\langle H, \xrightarrow{H}, \xleftarrow{H} \rangle$  of the database system, all the elements of  $H$  (the transactions) must be totally ordered by  $\xrightarrow{H}$ . This higher-level view of the database operations is implemented by lower-level operations that access individual database items. The higher-level system execution  $\langle H, \xrightarrow{H}, \xleftarrow{H} \rangle$  must be implemented by a lower-level one  $\langle S, \rightarrow, \dashrightarrow \rangle$  in which each transaction  $H$  in  $H$  is implemented by a set of lower-level operation executions in  $S$ .

Suppose  $G = \{G_1, \dots, G_m\}$  and  $H = \{H_1, \dots, H_n\}$  are elements of  $H$ , where the  $G_i$  and  $H_j$  are operation executions in  $S$ . For  $G \xrightarrow{H} H$  to hold, each  $G_i$  must precede ( $\rightarrow$ ) each  $H_j$ , and, conversely,  $H \xrightarrow{G} G$  only if each  $H_j$  precedes each  $G_i$ . In a situation like the one in Fig. 3, neither  $G \xrightarrow{H} H$  nor  $H \xrightarrow{G} G$  holds. (For a system with a global-time model, this means that both  $G \xrightarrow{H} H$  and  $H \xrightarrow{G} G$  hold.) If we required that the relations  $\xrightarrow{H}$  and  $\xleftarrow{H}$  be the induced relations  $\xrightarrow{H}$  and  $\xleftarrow{H}$ , then the only way to implement a serializable system, in which  $\xrightarrow{H}$  is a total ordering of the transactions, would be to prevent the type of interleaved execution shown in Fig. 3. The only allowable system executions would be those in which the transactions were actually executed serially - each transaction being completed before the next one is begun.

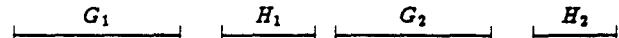


Fig. 3. An example with  $G \xrightarrow{H} H$  and  $H \xrightarrow{G} G$

Serial execution is, of course, too stringent a requirement because it prevents the concurrent execution of different transactions. We merely want to require that the system behave *as if* there were a serial execution. To show that a given system correctly implements a serializable database system, one specifies both the set of lower-level operation executions corresponding to each higher-level transaction and the precedence relation  $\xrightarrow{H}$  that describes the "as if" order, where the transactions act as if they had occurred in that order. This order must be consistent with the values read from the database - each read obtaining the value written by the most recent write of that item, where "most recent" is defined by  $\xrightarrow{H}$ .

As was observed in the introduction, the condition that a read obtain a value consistent with the ordering of the operations is not the only condition that must be placed upon  $\xrightarrow{H}$ . For the example in which each transaction either reads from or writes to the database, but does not do both, we must rule out an implementation that throws writes away and lets a read return the initial values of the database entries - an implementation that achieves serializability with a precedence relation  $\xrightarrow{H}$  in which all the read transactions precede all the write transactions. Although this implementation satisfies the requirement that every read obtain the most recently written value, this precedence relation is absurd because a read is defined to precede a write that may really have occurred years earlier.

Why is such a precedence relation absurd? In a real system, these database transactions may occur deep within the computer; we never actually see them happen. What is wrong with defining the precedence relation  $\xrightarrow{H}$  to pretend that these operation executions happened in any order we wish? After all, we are already pretending, contrary to fact, that the operations occur in some serial order.

In addition to reads and writes to database items, real systems perform externally observable operation executions such as printing on terminals. By observing these operation executions, we can infer precedence relations among the internal reads and writes. We need some condition on  $\xrightarrow{H}$  and  $\xleftarrow{H}$  to rule out pre-

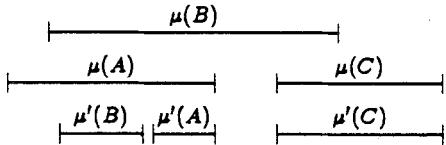


Fig. 4. An illustration of Proposition 1

cedence relations that contradict such observations.

It is shown below that these contradictions are avoided by requiring that if one higher-level operation execution “really” precedes another, then that precedence must appear in the “pretend” relations. Remembering that  $\rightarrow$  and  $\dashrightarrow$  are the “real” precedence relations and  $\xrightarrow{\mathcal{H}}$  and  $\xleftarrow{\mathcal{H}}$  are the “pretend” ones, this leads to the following definition.

**Definition 5.** A system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  implements a system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$  if  $\mathcal{H}$  is a higher-level view of  $\mathcal{S}$  and the following condition holds:

H3. For any  $G, H \in \mathcal{H}$ : if  $G \rightarrow H$  then  $G \xrightarrow{\mathcal{H}} H$ , where  $\rightarrow$  is defined by (2).

One justification for this definition in terms of global-time models is given by the following proposition, which is proved in [8]. (Recall that a global-time model is determined by the mapping  $\mu$ , since the set of events and their ordering is fixed.)

**Proposition 1.** Let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  and  $\langle \mathcal{S}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$  be system executions, both of which have global-time models, such that for any  $A, B \in \mathcal{S}$ :  $A \rightarrow B$  implies  $A \xrightarrow{\mathcal{H}} B$ . For any global-time model  $\mu$  of  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  there exists a global-time model  $\mu'$  of  $\langle \mathcal{S}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$  such that  $\mu'(A) \subseteq \mu(A)$  for every  $A$  in  $\mathcal{S}$ .

This proposition is illustrated in Fig. 4, where: (i)  $\mathcal{S} = \{A, B, C\}$ , (ii)  $A \rightarrow C$  is the only  $\rightarrow$  relation, and (iii)  $B \rightarrow A \rightarrow C$ . To apply Proposition 1 to Definition 5, substitute  $\mathcal{S}$  for  $\mathcal{H}$ , substitute  $\rightarrow$  and  $\dashrightarrow$  for  $\rightarrow$  and  $\dashrightarrow$ , and substitute  $\xrightarrow{\mathcal{H}}$  and  $\xleftarrow{\mathcal{H}}$  for  $\rightarrow$  and  $\dashrightarrow$ . The proposition then states that the “pretend” precedence relations are obtained from the real ones by shrinking the time interval during which the operation execution is considered to have occurred.

Let us return to the example of implementing a serializable database system. The formal requirement is that any system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ , whose operation executions consist

of reads and writes of individual database items, must implement a system  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$ , whose operations are database transactions, such that  $\xrightarrow{\mathcal{H}}$  is a total ordering of  $\mathcal{H}$ . By Proposition 1, this means that not only must the transactions be performed as if they had been executed in some sequential order, but that this order must be one that could have been obtained by executing each transaction within some interval of time during the period when it actually was executed. This rules out the absurd implementation described above, which implies a precedence relation  $\xrightarrow{\mathcal{H}}$  that makes writes come long after they actually occurred.

Another justification for Definition 5 is derived from the following result, which is proved in [8]. Its statement relies upon the obvious fact that if  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  is a system execution, then  $\langle \mathcal{T}, \rightarrow, \dashrightarrow \rangle$  is also a system execution for any subset  $\mathcal{T}$  of  $\mathcal{S}$ . (The symbols  $\rightarrow$  and  $\dashrightarrow$  denote both the relations on  $\mathcal{S}$  and their restrictions to  $\mathcal{T}$ . Also, in the proposition, the set  $\mathcal{T}$  is identified with the set of all singleton sets  $\{A\}$  for  $A \in \mathcal{T}$ .)

**Proposition 2.** Let  $\mathcal{S} \cup \mathcal{T}, \rightarrow, \dashrightarrow$  be a system execution, where  $\mathcal{S}$  and  $\mathcal{T}$  are disjoint; let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be an implementation of a system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$ ; and let  $\rightarrow$  and  $\dashrightarrow$  be the relations defined on  $\mathcal{H} \cup \mathcal{T}$  by (2). Then there exist precedence relations  $\xrightarrow{\mathcal{H} \cup \mathcal{T}}$  and  $\xleftarrow{\mathcal{H} \cup \mathcal{T}}$  such that:

- $\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{H} \cup \mathcal{T}}, \xleftarrow{\mathcal{H} \cup \mathcal{T}}$  is a system execution that is implemented by  $\mathcal{S} \cup \mathcal{T}, \rightarrow, \dashrightarrow$ .
- The restrictions of  $\xrightarrow{\mathcal{H} \cup \mathcal{T}}$  and  $\xleftarrow{\mathcal{H} \cup \mathcal{T}}$  to  $\mathcal{H}$  equal  $\xrightarrow{\mathcal{H}}$  and  $\xleftarrow{\mathcal{H}}$ , respectively.
- The restrictions of  $\xrightarrow{\mathcal{H} \cup \mathcal{T}}$  and  $\xleftarrow{\mathcal{H} \cup \mathcal{T}}$  to  $\mathcal{T}$  are extensions of the relations  $\rightarrow$  and  $\dashrightarrow$ , respectively.

To illustrate the significance of this proposition for Definition 5, let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be a system execution of reads and writes to database items that implements a higher-level system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xleftarrow{\mathcal{H}} \rangle$  of database transactions. The operation executions of  $\mathcal{S}$  presumably occur deep inside the computer and are not directly observable. Let  $\mathcal{T}$  be the set of all other operation executions in the system, including the externally observable ones. Proposition 2 means that, while the “pretend” precedence relations  $\xrightarrow{\mathcal{H}}$  and  $\xleftarrow{\mathcal{H}}$  may imply new precedence relations on the operation exe-

cutions in  $\mathcal{T}$ , these relations ( $\xrightarrow{\mathcal{H}}$  and  $\xrightarrow{\mathcal{H}, \mathcal{T}}$ ) are consistent with the “real” precedence relations  $\rightarrow$  and  $\dashv$  on  $\mathcal{T}$ . Thus, pretending that the database transactions occur in the order given by  $\xrightarrow{\mathcal{H}}$  does not contradict any of the real, externally observable orderings among the operations in  $\mathcal{T}$ .

When implementing a higher-level system, one usually ignores all operation executions that are not part of the implementation. For example, when implementing a database system, one considers only the transactions that access the database, ignoring the operation executions that initiate the transactions and use their results. This is justified by Proposition 2, which shows that the implementation cannot lead to any anomalous precedence relations among the operation executions that are being ignored.

A particularly simple kind of implementation is one in which each higher-level operation execution is implemented by a single lower-level one.

**Definition 6.** An implementation  $\langle \mathcal{S}, \rightarrow, \dashv \rangle$  of  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xrightarrow{\mathcal{H}, \mathcal{T}} \rangle$  is said to be *trivial* if every element of  $\mathcal{H}$  is a singleton set.

In a trivial implementation, the sets  $\mathcal{S}$  and  $\mathcal{H}$  are (essentially) the same; the two system executions differ only in their precedence relations. A trivial implementation is one that is not an implementation in the ordinary sense, but merely involves choosing new precedence relations (“as if” temporal relations).

### 3 Systems

A system execution has been defined, but not a system. Formally, a system is just a set of system executions – a set that represents all possible executions of the system.

**Definition 7.** A *system* is a set of system executions.

The usual method of describing a system is with a program written in some programming language. Each execution of such a program describes a system execution, and the program represents the system consisting of the set of all such executions. When considering communication and synchronization properties of concurrent systems, the only operation executions that are of interest are ones that involve interprocess communication – for example, the operations of sending a message or reading a

shared variable. Internal “calculation” steps can be ignored. If  $x$ ,  $y$ , and  $z$  are shared variables and  $a$  is local to the process in question, then an execution of the statement  $x := y + a * z$  includes three operation executions of interest: a read of  $y$ , a read of  $z$ , and a write of  $x$ . The actions of reading  $a$ , computing the product, and computing the sum are independent of the actions of other processes and could be considered to be either separate operation executions or part of the operation that writes the new value of  $x$ . For analyzing the interaction among processes, what is significant is that each of the two reads precedes ( $\rightarrow$ ) the write, and that no precedence relation is assumed between the two reads (assuming that the programming language does not specify an evaluation order within expressions).

A formal semantics for a programming language can be given by defining, for each syntactically correct program, the set of all possible executions. This is done by recursively defining a succession of lower and lower higher-level views, in which each operation execution represents a single execution of a syntactic program unit.<sup>3</sup> At the highest-level view, a system execution consists of a single operation execution that represents an execution of the entire program. A view in which an execution of the statement  $S; T$  is a single operation execution is refined into one in which an execution consists of an execution of  $S$  followed by ( $\rightarrow$ ) an execution of  $T$ .<sup>4</sup> While this kind of formal semantics may be useful in studying subtle programming language issues, it is unnecessary for the simple language constructs generally used in describing synchronization algorithms like the ones in Part II, so these ideas will just be employed informally.

Having defined what a system is, the next step is to define what it means for a system  $S$  to implement a higher-level system  $H$ . The higher-level system  $H$  can be regarded as a specification of the lower-level one  $S$ , so we must decide what it should mean for a system to meet a specification.

The system executions of  $S$  involve lower-level concepts such as program variables; those

<sup>3</sup> For nonterminating programs, the formalism must be extended to allow nonterminating higher-level operation executions, each one consisting of an infinite set of lower-level operation executions

<sup>4</sup> In the general case, we must also allow the possibility that an execution of  $S; T$  consists of a nonterminating execution of  $S$

of  $\mathbf{H}$  involve higher-level concepts such as transactions. The first thing we need is some way of interpreting a “concrete” system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  of the “real” implementation  $\mathbf{S}$  as an “abstract” execution of the “imaginary” high-level system  $\mathbf{H}$ . Thus, there must be some mapping  $\iota$  that assigns to any system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  of  $\mathbf{S}$  a higher-level system execution  $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$  that it implements. The implementation  $\mathbf{S}$ , which is a set of system executions, yields a set  $\iota(\mathbf{S})$  of higher-level system executions. What should be the relation between  $\iota(\mathbf{S})$  and  $\mathbf{H}$ ?

There are two distinct approaches to specification, which may be called the *prescriptive* and *restrictive* approaches. The prescriptive approach is generally employed by methods in which a system is specified with a high-level program, as in [9] and [11]. An implementation must be equivalent to the specification in the sense that it exhibits all the same possible behaviors as the specification. In the prescriptive approach, one requires that every possible execution of the specification  $\mathbf{H}$  be represented by some execution of  $\mathbf{S}$ , so  $\iota(\mathbf{S})$  must equal  $\mathbf{H}$ .

The restrictive approach is employed primarily by axiomatic methods, in which a system is specified by stating the properties it must satisfy. Any implementation that satisfies those properties is acceptable; it is not necessary for the implementation to allow all possible behaviors that satisfy the properties. If  $\mathbf{H}$  is the set of all system executions satisfying the required properties, then the restrictive approach requires only that every execution of  $\mathbf{S}$  represent some execution of  $\mathbf{H}$ , so  $\iota(\mathbf{S})$  must be contained in  $\mathbf{H}$ .

To illustrate the difference between the two approaches, consider the problem of implementing a program containing the statement  $x := y + a * z$  with a lower-level machine-language program. The statement does not specify in which order  $y$  and  $z$  are to be read, so  $\mathbf{H}$  should contain executions in which  $y$  is read before  $z$ , executions in which  $z$  is read before  $y$ , as well as ones in which they are read concurrently. With the prescriptive approach, a correct implementation would have to allow all of these possibilities, so a machine-language program that always reads  $y$  first then  $z$  would not be a correct implementation. In the restrictive approach, this is a perfectly acceptable implementation because it exhibits one of the allowed possibilities.

The usual reason for not specifying the or-

der of evaluation is to allow the compiler to choose any convenient order, not to require that it produce nondeterministic object code. I therefore find the restrictive approach to be the more natural and adopt it in the following definition.

*Definition 8.* The system  $\mathbf{S}$  implements a system  $\mathbf{H}$  if there is a mapping  $\iota: \mathbf{S} \rightarrow \mathbf{H}$  such that, for every system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  in  $\mathbf{S}$ ,  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  implements  $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$ .

In taking the restrictive approach, one faces the question of how to specify that the system must actually do anything. The specification of a banking system must allow a possible system execution in which no customers happen to use an automatic teller machine on a particular afternoon, and it must include the possibility that a customer will enter an invalid request. How can we rule out an implementation in which the machine simply ignores all customer requests during an afternoon, or interprets any request as an invalid one?

The answer lies in the concept of an *interface specification*, discussed in [7]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementer. The interface specification for the bank includes a description of what sequences of keystrokes at the teller machine constitute valid requests, and the set of system executions only includes ones in which every valid request is serviced. What it means for someone to use the machine is part of the interface specification, so the possibility of no one using the machine on some afternoon does not allow the implementation to ignore someone who does use it.

Part II considers only the internal operations that effect communication between processes within the system, not the interface operations that effect communication between the system and its environment. Therefore, the interface specification is not considered further. The reader is referred to [7] for a discussion of this subject.

## References

1. Bernstein PA, Goodman N (1981) Concurrency control in distributed database systems. ACM Comput Surv 13:185-222
2. Brauer W (ed) (1980) Net Theory and Applications. Lect Notes Comput Sci 84, Springer-Verlag, Berlin Heidelberg New York

3. Lamport L (in press) The mutual exclusion problem. *J ACM*
4. Lamport L (1979) A new approach to proving the correctness of multiprocess programs. *ACM Trans Program Lang Syst* 1:84-97
5. Lamport L. On interprocess communication. Part II: Algorithms. *Distributed Computing* 1:85-101
6. Lamport L (1978) Time, clocks and the ordering of events in a distributed system. *Commun ACM* 21:558-565
7. Lamport L (1985) What it means for a concurrent program to satisfy a specification: why no one has specified priority. In: *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, New Orleans
8. Lamport L (1985) Interprocess Communication. SRI Technical Report, March 1985
9. Lauer PE, Shields MW, Best E (1979) Formal Theory of the Basic COSY Notation. Technical Report TR143. Computing Laboratory, University of Newcastle upon Tyne
10. Mazurkiewicz A (1984) Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach. Technical Report 84-19, Institute of Applied Mathematics and Computer Science, University of Leiden
11. Milner R (1980) A Calculus of Communicating Systems. *Lect Notes Comput Sci* 92. Springer-Verlag, Berlin Heidelberg New York
12. Pnueli A (1977) The temporal logic of programs. In: *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, November 1977
13. Winskel G (1980) Events in Computation. PhD thesis, Edinburgh University

# **On Interprocess Communication**

Leslie Lamport

December 25, 1985

Much of this research was performed while the author was a member of the Computer Science Laboratory at SRI International, where it was sponsored by the Office of Naval Research Project under contract number N00014-84-C-0621 and the Rome Air Development Command Project under contract number F30602-85-C-0024.

### **Publication History**

The two parts of this report will appear as separate articles in *Distributed Computing*.

Copyright 1986 by Springer-Verlag. All rights reserved. Printed with permission.

## **Author's Abstract**

A formalism, not based upon atomic actions, for specifying and reasoning about concurrent systems is defined. It is used to specify several classes of interprocess communication mechanisms and to prove the correctness of algorithms for implementing them.

### **Capsule Review by Andrei Broder**

Concurrent systems are customarily described hierarchically, each level being intended to implement the level above it. On each level certain actions are considered atomic with respect to that level, although they decompose into a set of actions at a lower level. Furthermore there are cases when, for efficiency purposes, their components might be interleaved in time at a lower level with no loss of semantic correctness, despite the fact that the atomicity specified on the higher level is not respected. In this paper a very clean formalism is developed that allows a cohesive description of the different levels and axiomatic proofs of the implementation properties, without using the atomic action concept.

### **Capsule Review by Paul McJones**

A common approach to dealing with concurrency is to introduce primitives allowing the programmer to think in terms of the more familiar sequential model. For example, database transactions and linguistic constructs for mutual exclusion such as the monitor give a process the illusion that there is no concurrency. In contrast, Part II of this paper presents the approach of designing and verifying algorithms that work in the face of manifest concurrency.

Starting from some seemingly minimal assumptions about the nature of communication between asynchronous processes, the author proposes a classification of twelve partially-ordered kinds of single-writer shared registers. He provides constructions for implementing many of these classes from “weaker” ones, culminating in a multi-value, single-reader, atomic register. The constructions are proved both informally and using the formalism of Part I.

Much of the paper is of a theoretical nature. However, its ideas are worth study by system builders. For example, its algorithms and verification techniques could be of use in designing a “conventional” synchronization mechanism (e.g. a semaphore) for a multiprocessor system. A more exciting possibility would be to extend its approach to the design of a higher level concurrent algorithm such as taking a snapshot of an online database.



## Contents

|                                                   |           |
|---------------------------------------------------|-----------|
| <b>I Basic Formalism</b>                          | <b>1</b>  |
| <b>1 System Executions</b>                        | <b>2</b>  |
| <b>2 Hierarchical Views</b>                       | <b>6</b>  |
| <b>3 Systems</b>                                  | <b>12</b> |
| <br>                                              |           |
| <b>II Algorithms</b>                              | <b>17</b> |
| <b>4 The Nature of Asynchronous Communication</b> | <b>17</b> |
| <b>5 The Constructions</b>                        | <b>22</b> |
| <b>6 Register Axioms</b>                          | <b>31</b> |
| <b>7 Correctness Proofs for the Constructions</b> | <b>38</b> |
| 7.1 Proof of Constructions 1, 2, and 3 . . . . .  | 38        |
| 7.2 Proof of Construction 4 . . . . .             | 40        |
| 7.3 Proof of Construction 5 . . . . .             | 42        |
| <b>8 Conclusion</b>                               | <b>45</b> |



# Part I

## Basic Formalism

This paper addresses what I believe to be fundamental questions in the theory of interprocess communication. Part I develops a formal definition of what it means to implement one system with a lower-level one and provides a method for reasoning about concurrent systems. The definitions and axioms introduced here are applied in Part II to algorithms that implement certain interprocess communication mechanisms. Readers interested only in these mechanisms and not in the formalism can skip Part I and read only Sections 4 and 5 of Part II.

To motivate the formalism, let us consider the question of atomicity. Most treatments of concurrent processing assume the existence of atomic operations—an atomic operation being one whose execution is performed as an indivisible action. The term *operation* is used to mean a class of actions such as depositing money in a bank account, and the term *operation execution* to mean one specific instance of executing such an action—for example, depositing \$100 in account number 14335 at 10:35AM on December 14, 1987. Atomic operations must be implemented in terms of lower-level operations. A high-level language may provide a *P* operation to a semaphore as an atomic operation, but this operation must be implemented in terms of lower-level machine-language instructions. Viewed at the machine-language level, the semaphore operation is not atomic. Moreover, the machine-language operations must ultimately be implemented with circuits in which operations are manifestly nonatomic—the possibility of harmful “race conditions” shows that the setting and the testing of a flip-flop are not atomic actions.

Part II considers the problem of implementing atomic operations to a shared register with more primitive, nonatomic operations. Here, a more familiar example of implementing atomicity is used: concurrency control in a database. In a database system, higher-level transactions, which may read and modify many individual data items, are implemented with lower-level reads and writes of single items. These lower-level read and write operations are assumed to be atomic, and the problem is to make the higher-level transactions atomic. It is customary to say that a semaphore operation *is* atomic while a database transaction *appears to be* atomic, but this verbal distinction has no fundamental significance.

In database systems, atomicity of transactions is achieved by implementing a *serializable* execution order. The lower-level accesses performed by the

different transactions are scheduled so that the net effect is the same as if the transactions had been executed in some serial order—first executing all the lower-level accesses comprising one transaction, then executing all the accesses of the next transaction, and so on. The transactions should not actually be scheduled in such a serial fashion, since this would be inefficient; it is necessary only that the effect be the same as if that were done.<sup>1</sup>

In the literature on concurrency control in databases, serializability is usually the only correctness condition that is stated [1]. However, serializability by itself does not ensure correctness. Consider a database system in which each transaction either reads from or writes to the database, but does not do both. Moreover, assume that the system has a finite lifetime, at the end of which it is to be scrapped. Serializability is achieved by an implementation in which reads always return the initial value of the database entries and writes are simply not executed. This yields the same results as a serial execution in which one first performs all the read transactions and then all the writes. While such an implementation satisfies the requirement of serializability, no one would consider it to be correct.

This example illustrates the need for a careful examination of what it means for one system to implement another. It is reconsidered in Section 2, where the additional correctness condition needed to rule out this absurd implementation is stated.

## 1 System Executions

Almost all models of concurrent processes have indivisible atomic actions as primitive elements. For example, models in which a process is represented by a sequence or “trace” [11, 15, 16] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [10, 12] assume that the firing of an individual transition is atomic. These models are not appropriate for studying such fundamental questions as what it means to implement an atomic operation, in which the nonatomicity of operations must be directly addressed.

More conventional formalisms are therefore eschewed in favor of one introduced in [7] and refined in [6], in which the primitive elements are

---

<sup>1</sup>In the context of databases, atomicity often denotes the additional property that a failure cannot leave the database in a state reflecting a partially completed transaction. In this paper, the possibility of failure is ignored, so no distinction between atomicity and serializability is made.

*operation executions* that are not assumed to be atomic. This formalism is described below; the reader is referred to [7] and [6] for more details.

A *system execution* consists of a set of *operation executions*, together with certain temporal precedence relations on these operation executions. Recall that an operation execution represents a single execution of some operation. When all operations are assumed to be atomic, an operation execution  $A$  can influence another operation execution  $B$  only if  $A$  precedes  $B$ —meaning that all actions of  $A$  are completed before any action of  $B$  is begun. In this case, one needs only a single temporal relation  $\rightarrow$ , read “precedes”, to describe the temporal ordering among operation executions. While temporal precedence is usually considered to be a total ordering of atomic operations, in distributed systems it is best thought of as an irreflexive partial ordering (see [8]).

Nonatomicity introduces the possibility that an operation execution  $A$  can influence an operation execution  $B$  without preceding it; it is necessary only that some action of  $A$  precede some action of  $B$ . Hence, in addition to the precedence relation  $\rightarrow$ , one needs an additional relation  $\dashrightarrow$ , read “can affect”, where  $A \dashrightarrow B$  means that some action of  $A$  precedes some action of  $B$ .

**Definition 1** A system execution is a triple  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ , where  $\mathcal{S}$  is a finite or countably infinite set whose elements are called operation executions, and  $\rightarrow$  and  $\dashrightarrow$  are precedence relations on  $\mathcal{S}$  satisfying axioms A1–A5 below.

To assist in understanding the axioms for the  $\rightarrow$  and  $\dashrightarrow$  relations, it is helpful to have a semantic model for the formalism. The model to be used is one in which an operation execution is represented by a set of primitive actions or events, where  $A \rightarrow B$  means that all the events of  $A$  precede all the events of  $B$ , and  $A \dashrightarrow B$  means that some event of  $A$  precedes some event of  $B$ . Letting  $\mathbf{E}$  denote the set of all events, and  $\rightarrow$  the temporal precedence relation among events, we get the following formal definition.

**Definition 2** A model of a system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  consists of a triple  $\langle \mathbf{E}, \rightarrow, \mu \rangle$ , where  $\mathbf{E}$  is a set,  $\rightarrow$  is an irreflexive partial ordering on  $\mathbf{E}$ , and  $\mu$  is a mapping that assigns to each operation execution  $A$  of  $\mathcal{S}$  a nonempty subset  $\mu(A)$  of  $\mathbf{E}$ , such that for every pair of operation executions  $A$  and  $B$  of  $\mathcal{S}$ :

$$\begin{aligned} A \rightarrow B &\equiv \forall a \in \mu(A) : \forall b \in \mu(B) : a \rightarrow b \\ A \dashrightarrow B &\equiv \exists a \in \mu(A) : \exists b \in \mu(B) : a \rightarrow b \text{ or } a = b \end{aligned} \quad (1)$$

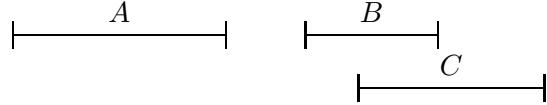


Figure 1: Three operation executions in a global-time model.

Note that the same symbol  $\longrightarrow$  denotes the “precedes” relation both between operation executions in  $\mathcal{S}$  and between events in  $\mathbf{E}$ .

Other than the existence of the temporal partial-ordering relation  $\longrightarrow$ , no assumption is made about the structure of the set of events  $\mathbf{E}$ . In particular, operation executions may be modeled as infinite sets of events. An important class of models is obtained by letting  $\mathbf{E}$  be the set of events in four-dimensional spacetime, with  $\longrightarrow$  the “happens before” relation of special relativity, where  $a \longrightarrow b$  means that it is temporally possible for event  $a$  to causally affect event  $b$ .

Another simple and useful class of models is obtained by letting  $\mathbf{E}$  be the real number line and representing each operation execution  $A$  as a closed interval.

**Definition 3** A global-time model of a system execution  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  is one in which  $\mathbf{E}$  is the set of real numbers,  $\longrightarrow$  is the ordinary  $<$  relation, and each set  $\mu(A)$  is of the form  $[s_A, f_A]$  with  $s_A < f_A$ .

Think of  $s_A$  and  $f_A$  as the starting and finishing times of  $A$ . In a global-time model,  $A \longrightarrow B$  means that  $A$  finishes before  $B$  starts, and  $A \dashrightarrow B$  means that  $A$  starts before (or at the same time as)  $B$  finishes. These relations are illustrated by Figure 1, where operation executions  $A$ ,  $B$ , and  $C$ , represented by the three indicated intervals, satisfy:  $A \longrightarrow B$ ,  $A \longrightarrow C$ ,  $B \dashrightarrow C$ , and  $C \dashrightarrow B$ . (In this and similar figures, the number line runs from left to right, and overlapping intervals are drawn one above the other.)

To complete Definition 1, the axioms for the precedence relations  $\longrightarrow$  and  $\dashrightarrow$  of a system execution must be given. They are the following, where  $A$ ,  $B$ ,  $C$ , and  $D$  denote arbitrary operation executions in  $\mathcal{S}$ . Axiom A4 is illustrated (in a global-time model) by Figure 2; the reader is urged to draw similar pictures to help understand the other axioms.

A1. The relation  $\longrightarrow$  is an irreflexive partial ordering.

A2. If  $A \longrightarrow B$  then  $A \dashrightarrow B$  and  $B \not\rightarrow A$ .

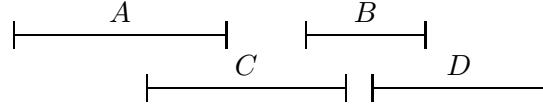


Figure 2: An illustration of Axiom A4.

A3. If  $A \rightarrow B \dashrightarrow C$  or  $A \dashrightarrow B \rightarrow C$  then  $A \dashrightarrow C$ .

A4. If  $A \rightarrow B \dashrightarrow C \rightarrow D$  then  $A \rightarrow D$ .

A5. For any  $A$ , the set of all  $B$  such that  $A \not\rightarrow B$  is finite.

(These axioms differ from the ones in [6] because only terminating operation executions are considered here.)

Axioms A1–A4 follow from (1), so they do not constrain the choice of a model. Axiom A5 does not follow from (1); it restricts the class of allowed models. Intuitively, A5 asserts that a system execution begins at some point in time, rather than extending into the infinite past. When  $\mathbf{E}$  is the set of events in space-time, A5 holds for any model in which: (i) each operation occupies a finite region of space-time, (ii) any finite region of space-time contains only a finite number of operation executions, and (iii) the system is not expanding faster than the speed of light.<sup>2</sup>

Most readers will find it easiest to think about system executions in terms of a global-time model, and to interpret the relations  $\rightarrow$  and  $\dashrightarrow$  as indicated by the example in Figure 1. Such a mental model is adequate for most purposes. However, the reader should be aware that in a system execution having a global-time model, for any distinct operation executions  $A$  and  $B$ , either  $A \rightarrow B$  or  $B \dashrightarrow A$ . (In fact, this is a necessary and sufficient condition for a system execution to have a global-time model [5].) However, in a system execution without a global-time model, it is possible for neither  $A \rightarrow B$  nor  $B \dashrightarrow A$  to hold. As a trivial counterexample, let  $\mathcal{S}$  consist of two elements and let the relations  $\rightarrow$  and  $\dashrightarrow$  be empty.

While a global-time model is a valuable aid to acquiring an intuitive understanding of a system, it is better to use more abstract reasoning when proving properties of systems. The relations  $\rightarrow$  and  $\dashrightarrow$  capture the essential temporal properties of a system execution, and A1–A5 provide the

---

<sup>2</sup>A system expanding faster than the speed of light could have an infinite number of operation executions none of which are preceded by any operation.

necessary tools for reasoning about these relations. It has been my experience that proofs based upon these axioms are simpler and more instructive than ones that involve modeling operation executions as sets of events.

## 2 Hierarchical Views

A system can be viewed at different levels of detail, with different operation executions at each level. Viewed at the customer's level, a banking system has operation executions such as *deposit* \$1000. Viewed at the programmer's level, this same system executes operations such as *dep\_amt[cust]* := 1000. The fundamental problem of system building is to implement one system (like a banking system) as a higher-level view of another system (like a Pascal program).

A higher-level operation consists of a set of lower-level operations—the set of operations that implement it. Let  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  be a system execution and let  $\mathcal{H}$  be a set whose elements, called *higher-level operation executions*, are sets of operation executions from  $\mathcal{S}$ . A model for  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  represents each operation execution in  $\mathcal{S}$  by a set of events. This gives a representation of each higher-level operation execution  $H$  in  $\mathcal{H}$  as a set of events—namely, the set of all events contained in the representation of the lower-level operation executions that comprise  $H$ . This in turn defines precedence relations  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$ , where  $G \xrightarrow{*} H$  means that all events in (the representation of)  $G$  precede all events in  $H$ , and  $G \dashrightarrow^{*} H$  means that some event in  $G$  precedes some event in  $H$ , for  $G$  and  $H$  in  $\mathcal{H}$ .

To express all this formally, let  $\mathbf{E}, \longrightarrow, \mu$  be a model for  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ , define the mapping  $\mu^*$  on  $\mathcal{H}$  by

$$\mu^*(H) = \bigcup\{\mu(A) : A \in H\}$$

and define the precedence relations  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$  on  $\mathcal{H}$  by

$$\begin{aligned} G \xrightarrow{*} H &\equiv \forall g \in \mu^*(G) : \forall h \in \mu^*(H) : g \longrightarrow h \\ G \dashrightarrow^{*} H &\equiv \exists g \in \mu^*(G) : \exists h \in \mu^*(H) : g \longrightarrow h \text{ or } g = h \end{aligned}$$

Using (1), it is easy to show that these precedence relations are the same ones obtained by the following definitions:

$$\begin{aligned} G \xrightarrow{*} H &\equiv \forall A \in G : \forall B \in H : A \longrightarrow B \\ G \dashrightarrow^{*} H &\equiv \exists A \in G : \exists B \in H : A \dashrightarrow B \text{ or } A = B \end{aligned} \tag{2}$$

Observe that  $\xrightarrow{*}$  and  $\dashv^*$  are expressed directly in terms of the  $\rightarrow$  and  $\dashv$  relations on  $\mathcal{S}$ , without reference to any model. We take (2) to be the definition of the relations  $\xrightarrow{*}$  and  $\dashv^*$ .

For the triple  $\langle \mathcal{H}, \xrightarrow{*}, \dashv^* \rangle$  to be a system execution, the relations  $\xrightarrow{*}$  and  $\dashv^*$  must satisfy axioms A1–A5. If each element of  $\mathcal{H}$  is assumed to be a nonempty set of operation executions, then Axioms A1–A4 follow from (2) and the corresponding axioms for  $\rightarrow$  and  $\dashv$ . For A5 to hold, it is sufficient that each element of  $\mathcal{H}$  consist of a finite number of elements of  $\mathcal{S}$ , and that each element of  $\mathcal{S}$  belong to a finite number of elements of  $\mathcal{H}$ . Adding the natural requirement that every lower-level operation execution be part of some higher-level one, this leads to the following definition.

**Definition 4** *A higher-level view of a system execution  $\langle \mathcal{S}, \rightarrow, \dashv \rangle$  consists of a set  $\mathcal{H}$  such that:*

- H1. Each element of  $\mathcal{H}$  is a finite, nonempty set of elements of  $\mathcal{S}$ .*
- H2. Each element of  $\mathcal{S}$  belongs to a finite, nonzero number of elements of  $\mathcal{H}$ .*

In most cases of interest,  $\mathcal{H}$  is a partition of  $\mathcal{S}$ , so each element of  $\mathcal{S}$  belongs to exactly one element of  $\mathcal{H}$ . However, Definition 4 allows the more general case in which a single lower-level operation execution is viewed as part of the implementation of more than one higher-level one.

Let us now consider what it should mean for one system to implement another. If the system execution  $\langle \mathcal{S}, \rightarrow, \dashv \rangle$  is an implementation of a system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashv^{\mathcal{H}} \rangle$ , then we expect  $\mathcal{H}$  to be a higher-level view of  $\mathcal{S}$ —that is, each operation in  $\mathcal{H}$  should consist of a set of operation executions of  $\mathcal{S}$  satisfying H1 and H2. This describes the elements of  $\mathcal{H}$ , but not the precedence relations  $\xrightarrow{\mathcal{H}}$  and  $\dashv^{\mathcal{H}}$ . What should those relations be?

If we consider the operation executions in  $\mathcal{S}$  to be the “real” ones, and the elements of  $\mathcal{H}$  to be fictitious groupings of the real operation executions into abstract, higher-level ones, then the induced precedence relations  $\xrightarrow{*}$  and  $\dashv^*$  represent the “real” temporal relations on  $\mathcal{H}$ . These induced relations make the higher-level view  $\mathcal{H}$  a system execution, so they are an obvious choice for the relations  $\xrightarrow{\mathcal{H}}$  and  $\dashv^{\mathcal{H}}$ . However, as we shall see, they may not be the proper choice.

Let us return to the problem of implementing atomic database operations. Atomicity requires that, when viewed at the level at which the

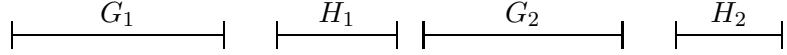


Figure 3: An example with  $G \not\rightarrow H$  and  $H \not\rightarrow G$ .

operation executions are the transactions, the transactions appear to be executed sequentially. In terms of our formalism, the correctness condition is that, in any system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow \rangle$  of the database system, all the elements of  $\mathcal{H}$  (the transactions) must be totally ordered by  $\xrightarrow{\mathcal{H}}$ . This higher-level view of the database operations is implemented by lower-level operations that access individual database items. The higher-level system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow \rangle$  must be implemented by a lower-level one  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  in which each transaction  $H$  in  $\mathcal{H}$  is implemented by a set of lower-level operation executions in  $\mathcal{S}$ .

Suppose  $G = \{G_1, \dots, G_m\}$  and  $H = \{H_1, \dots, H_n\}$  are elements of  $\mathcal{H}$ , where the  $G_i$  and  $H_i$  are operation executions in  $\mathcal{S}$ . For  $G \xrightarrow{*} H$  to hold, each  $G_i$  must precede ( $\rightarrow$ ) each  $H_j$ , and, conversely,  $H \xrightarrow{*} G$  only if each  $H_j$  precedes each  $G_i$ . In a situation like the one in Figure 3, neither  $G \xrightarrow{*} H$  nor  $H \xrightarrow{*} G$  holds. (For a system with a global-time model, this means that both  $G \dashrightarrow^* H$  and  $H \dashrightarrow^* G$  hold.) If we required that the relations  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$  be the induced relations  $\xrightarrow{*}$  and  $\dashrightarrow^*$ , then the only way to implement a serializable system, in which  $\xrightarrow{\mathcal{H}}$  is a total ordering of the transactions, would be to prevent the type of interleaved execution shown in Figure 3. The only allowable system executions would be those in which the transactions were actually executed serially—each transaction being completed before the next one is begun.

Serial execution is, of course, too stringent a requirement because it prevents the concurrent execution of different transactions. We merely want to require that the system behave *as if* there were a serial execution. To show that a given system correctly implements a serializable database system, one specifies both the set of lower-level operation executions corresponding to each higher-level transaction and the precedence relation  $\xrightarrow{\mathcal{H}}$  that describes the “as if” order, where the transactions act as if they had occurred in that order. This order must be consistent with the values read from the database—each read obtaining the value written by the most recent write of that item, where “most recent” is defined by  $\xrightarrow{\mathcal{H}}$ .

As was observed in the introduction, the condition that a read obtain a

value consistent with the ordering of the operations is not the only condition that must be placed upon  $\xrightarrow{\mathcal{H}}$ . For the example in which each transaction either reads from or writes to the database, but does not do both, we must rule out an implementation that throws writes away and lets a read return the initial values of the database entries—an implementation that achieves serializability with a precedence relation  $\xrightarrow{\mathcal{H}}$  in which all the read transactions precede all the write transactions. Although this implementation satisfies the requirement that every read obtain the most recently written value, this precedence relation is absurd because a read is defined to precede a write that may really have occurred years earlier.

Why is such a precedence relation absurd? In a real system, these database transactions may occur deep within the computer; we never actually see them happen. What is wrong with defining the precedence relation  $\xrightarrow{\mathcal{H}}$  to pretend that these operation executions happened in any order we wish? After all, we are already pretending, contrary to fact, that the operations occur in some serial order.

In addition to reads and writes to database items, real systems perform externally observable operation executions such as printing on terminals. By observing these operation executions, we can infer precedence relations among the internal reads and writes. We need some condition on  $\xrightarrow{\mathcal{H}}$  and  $\xrightarrow{\mathcal{H}}$  to rule out precedence relations that contradict such observations.

It is shown below that these contradictions are avoided by requiring that if one higher-level operation execution “really” precedes another, then that precedence must appear in the “pretend” relations. Remembering that  $\xrightarrow{*}$  and  $\xrightarrow{-*}$  are the “real” precedence relations and  $\xrightarrow{\mathcal{H}}$  and  $\xrightarrow{-\mathcal{H}}$  are the “pretend” ones, this leads to the following definition.

**Definition 5** *A system execution  $\langle \mathcal{S}, \xrightarrow{*}, \xrightarrow{-*} \rangle$  implements a system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \xrightarrow{-\mathcal{H}} \rangle$  if  $\mathcal{H}$  is a higher-level view of  $\mathcal{S}$  and the following condition holds:*

*H3. For any  $G, H \in \mathcal{H}$ : if  $G \xrightarrow{*} H$  then  $G \xrightarrow{\mathcal{H}} H$ , where  $\xrightarrow{*}$  is defined by (2).*

One justification for this definition in terms of global-time models is given by the following proposition, which is proved in [5]. (Recall that a global-time model is determined by the mapping  $\mu$ , since the set of events and their ordering is fixed.)

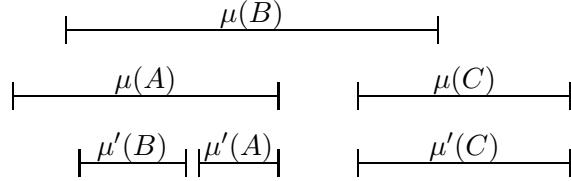


Figure 4: An illustration of Proposition 1.

**Proposition 1** *Let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  and  $\langle \mathcal{S}, \stackrel{*}{\rightarrow}, \stackrel{*}{\dashrightarrow} \rangle$  be system executions, both of which have global-time models, such that for any  $A, B \in \mathcal{S}$ :  $A \rightarrow B$  implies  $A \stackrel{*}{\rightarrow} B$ . For any global-time model  $\mu$  of  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  there exists a global-time model  $\mu'$  of  $\langle \mathcal{S}, \stackrel{*}{\rightarrow}, \stackrel{*}{\dashrightarrow} \rangle$  such that  $\mu'(A) \subseteq \mu(A)$  for every  $A$  in  $\mathcal{S}$ .*

This proposition is illustrated in Figure 4, where: (i)  $\mathcal{S} = \{A, B, C\}$ , (ii)  $A \rightarrow C$  is the only  $\rightarrow$  relation, and (iii)  $B \stackrel{*}{\rightarrow} A \stackrel{*}{\rightarrow} C$ . To apply Proposition 1 to Definition 5, substitute  $\mathcal{S}$  for  $\mathcal{H}$ , substitute  $\stackrel{*}{\rightarrow}$  and  $\stackrel{*}{\dashrightarrow}$  for  $\rightarrow$  and  $\dashrightarrow$ , and substitute  $\stackrel{\mathcal{H}}{\rightarrow}$  and  $\stackrel{\mathcal{H}}{\dashrightarrow}$  for  $\stackrel{*}{\rightarrow}$  and  $\stackrel{*}{\dashrightarrow}$ . The proposition then states that the “pretend” precedence relations are obtained from the real ones by shrinking the time interval during which the operation execution is considered to have occurred.

Let us return to the example of implementing a serializable database system. The formal requirement is that any system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ , whose operation executions consist of reads and writes of individual database items, must implement a system  $\langle \mathcal{H}, \stackrel{\mathcal{H}}{\rightarrow}, \stackrel{\mathcal{H}}{\dashrightarrow} \rangle$ , whose operations are database transactions, such that  $\stackrel{\mathcal{H}}{\rightarrow}$  is a total ordering of  $\mathcal{H}$ . By Proposition 1, this means that not only must the transactions be performed as if they had been executed in some sequential order, but that this order must be one that could have been obtained by executing each transaction within some interval of time during the period when it actually was executed. This rules out the absurd implementation described above, which implies a precedence relation  $\stackrel{\mathcal{H}}{\rightarrow}$  that makes writes come long after they actually occurred.

Another justification for Definition 5 is derived from the following result, which is proved in [5]. Its statement relies upon the obvious fact that if  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  is a system execution, then  $\langle \mathcal{T}, \rightarrow, \dashrightarrow \rangle$  is also a system execution for any subset  $\mathcal{T}$  of  $\mathcal{S}$ . (The symbols  $\rightarrow$  and  $\dashrightarrow$  denote both the relations on  $\mathcal{S}$  and their restrictions to  $\mathcal{T}$ . Also, in the proposition, the set  $\mathcal{T}$  is identified with the set of all singleton sets  $\{A\}$  for  $A \in \mathcal{T}$ .)

**Proposition 2** Let  $\mathcal{S} \cup \mathcal{T}, \rightarrow, \dashrightarrow$  be a system execution, where  $\mathcal{S}$  and  $\mathcal{T}$  are disjoint; let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be an implementation of a system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ ; and let  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$  be the relations defined on  $\mathcal{H} \cup \mathcal{T}$  by (2). Then there exist precedence relations  $\xrightarrow{\mathcal{HT}}$  and  $\dashrightarrow^{\mathcal{HT}}$  such that:

- $\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{HT}}, \dashrightarrow^{\mathcal{HT}}$  is a system execution that is implemented by  $\mathcal{S} \cup \mathcal{T}, \rightarrow, \dashrightarrow$ .
- The restrictions of  $\xrightarrow{\mathcal{HT}}$  and  $\dashrightarrow^{\mathcal{HT}}$  to  $\mathcal{H}$  equal  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$ , respectively.
- The restrictions of  $\xrightarrow{\mathcal{HT}}$  and  $\dashrightarrow^{\mathcal{HT}}$  to  $\mathcal{T}$  are extensions of the relations  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$ , respectively.

To illustrate the significance of this proposition for Definition 5, let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be a system execution of reads and writes to database items that implements a higher-level system execution  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$  of database transactions. The operation executions of  $\mathcal{S}$  presumably occur deep inside the computer and are not directly observable. Let  $\mathcal{T}$  be the set of all other operation executions in the system, including the externally observable ones. Proposition 2 means that, while the “pretend” precedence relations  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$  may imply new precedence relations on the operation executions in  $\mathcal{T}$ , these relations ( $\xrightarrow{\mathcal{HT}}$  and  $\dashrightarrow^{\mathcal{HT}}$ ) are consistent with the “real” precedence relations  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$  on  $\mathcal{T}$ . Thus, pretending that the database transactions occur in the order given by  $\xrightarrow{\mathcal{H}}$  does not contradict any of the real, externally observable orderings among the operations in  $\mathcal{T}$ .

When implementing a higher-level system, one usually ignores all operation executions that are not part of the implementation. For example, when implementing a database system, one considers only the transactions that access the database, ignoring the operation executions that initiate the transactions and use their results. This is justified by Proposition 2, which shows that the implementation cannot lead to any anomalous precedence relations among the operation executions that are being ignored.

A particularly simple kind of implementation is one in which each higher-level operation execution is implemented by a single lower-level one.

**Definition 6** An implementation  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  of  $\langle \mathcal{H}, \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$  is said to be trivial if every element of  $\mathcal{H}$  is a singleton set.

In a trivial implementation, the sets  $\mathcal{S}$  and  $\mathcal{H}$  are (essentially) the same; the two system executions differ only in their precedence relations. A trivial implementation is one that is not an implementation in the ordinary sense, but merely involves choosing new precedence relations (“as if” temporal relations).

### 3 Systems

A system execution has been defined, but not a system. Formally, a system is just a set of system executions—a set that represents all possible executions of the system.

**Definition 7** *A system is a set of system executions.*

The usual method of describing a system is with a program written in some programming language. Each execution of such a program describes a system execution, and the program represents the system consisting of the set of all such executions. When considering communication and synchronization properties of concurrent systems, the only operation executions that are of interest are ones that involve interprocess communication—for example, the operations of sending a message or reading a shared variable. Internal “calculation” steps can be ignored. If  $x$ ,  $y$ , and  $z$  are shared variables and  $a$  is local to the process in question, then an execution of the statement  $x := y + a * z$  includes three operation executions of interest: a read of  $y$ , a read of  $z$ , and a write of  $x$ . The actions of reading  $a$ , computing the product, and computing the sum are independent of the actions of other processes and could be considered to be either separate operation executions or part of the operation that writes the new value of  $x$ . For analyzing the interaction among processes, what is significant is that each of the two reads precedes ( $\rightarrow$ ) the write, and that no precedence relation is assumed between the two reads (assuming that the programming language does not specify an evaluation order within expressions).

A formal semantics for a programming language can be given by defining, for each syntactically correct program, the set of all possible executions. This is done by recursively defining a succession of lower and lower higher-level views, in which each operation execution represents a single execution of a syntactic program unit.<sup>3</sup> At the highest-level view, a system execution

---

<sup>3</sup>For nonterminating programs, the formalism must be extended to allow nonterminating higher-level operation executions, each one consisting of an infinite set of lower-level

consists of a single operation execution that represents an execution of the entire program. A view in which an execution of the statement  $S; T$  is a single operation execution is refined into one in which an execution consists of an execution of  $S$  followed by ( $\longrightarrow$ ) an execution of  $T$ .<sup>4</sup> While this kind of formal semantics may be useful in studying subtle programming language issues, it is unnecessary for the simple language constructs generally used in describing synchronization algorithms like the ones in Part II, so these ideas will just be employed informally.

Having defined what a system is, the next step is to define what it means for a system  $\mathbf{S}$  to implement a higher-level system  $\mathbf{H}$ . The higher-level system  $\mathbf{H}$  can be regarded as a specification of the lower-level one  $\mathbf{S}$ , so we must decide what it should mean for a system to meet a specification.

The system executions of  $\mathbf{S}$  involve lower-level concepts such as program variables; those of  $\mathbf{H}$  involve higher-level concepts such as transactions. The first thing we need is some way of interpreting a “concrete” system execution  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  of the “real” implementation  $\mathbf{S}$  as an “abstract” execution of the “imaginary” high-level system  $\mathbf{H}$ . Thus, there must be some mapping  $\iota$  that assigns to any system execution  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  of  $\mathbf{S}$  a higher-level system execution  $\iota(\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle)$  that it implements. The implementation  $\mathbf{S}$ , which is a set of system executions, yields a set  $\iota(\mathbf{S})$  of higher-level system executions. What should be the relation between  $\iota(\mathbf{S})$  and  $\mathbf{H}$ ?

There are two distinct approaches to specification, which may be called the *prescriptive* and *restrictive* approaches. The prescriptive approach is generally employed by methods in which a system is specified with a high-level program, as in [10] and [12]. An implementation must be equivalent to the specification in the sense that it exhibits all the same possible behaviors as the specification. In the prescriptive approach, one requires that every possible execution of the specification  $\mathbf{H}$  be represented by some execution of  $\mathbf{S}$ , so  $\iota(\mathbf{S})$  must equal  $\mathbf{H}$ .

The restrictive approach is employed primarily by axiomatic methods, in which a system is specified by stating the properties it must satisfy. Any implementation that satisfies those properties is acceptable; it is not necessary for the implementation to allow all possible behaviors that satisfy the properties. If  $\mathbf{H}$  is the set of all system executions satisfying the required properties, then the restrictive approach requires only that every execution

---

operation executions.

<sup>4</sup>In the general case, we must also allow the possibility that an execution of  $S; T$  consists of a nonterminating execution of  $S$ .

of  $\mathbf{S}$  represent some execution of  $\mathbf{H}$ , so  $\iota(\mathbf{S})$  must be contained in  $\mathbf{H}$ .

To illustrate the difference between the two approaches, consider the problem of implementing a program containing the statement  $x := y + a * z$  with a lower-level machine-language program. The statement does not specify in which order  $y$  and  $z$  are to be read, so  $\mathbf{H}$  should contain executions in which  $y$  is read before  $z$ , executions in which  $z$  is read before  $y$ , as well as ones in which they are read concurrently. With the prescriptive approach, a correct implementation would have to allow all of these possibilities, so a machine-language program that always reads  $y$  first then  $z$  would not be a correct implementation. In the restrictive approach, this is a perfectly acceptable implementation because it exhibits one of the allowed possibilities.

The usual reason for not specifying the order of evaluation is to allow the compiler to choose any convenient order, not to require that it produce nondeterministic object code. I therefore find the restrictive approach to be the more natural and adopt it in the following definition.

**Definition 8** *The system  $\mathbf{S}$  implements a system  $\mathbf{H}$  if there is a mapping  $\iota : \mathbf{S} \mapsto \mathbf{H}$  such that, for every system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  in  $\mathbf{S}$ ,  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  implements  $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$ .*

In taking the restrictive approach, one faces the question of how to specify that the system must actually do anything. The specification of a banking system must allow a possible system execution in which no customers happen to use an automatic teller machine on a particular afternoon, and it must include the possibility that a customer will enter an invalid request. How can we rule out an implementation in which the machine simply ignores all customer requests during an afternoon, or interprets any request as an invalid one?

The answer lies in the concept of an *interface specification*, discussed in [9]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementor. The interface specification for the bank includes a description of what sequences of keystrokes at the teller machine constitute valid requests, and the set of system executions only includes ones in which every valid request is serviced. What it means for someone to use the machine is part of the interface specification, so the possibility of no one using the machine on some afternoon does not allow the implementation to ignore someone who does use it.

Part II considers only the internal operations that effect communication between processes within the system, not the interface operations that effect

communication between the system and its environment. Therefore, the interface specification is not considered further. The reader is referred to [9] for a discussion of this subject.



## Part II

# Algorithms

Part I describes a formalism for specifying and reasoning about concurrent systems. Here in Part II, communication between asynchronous processes in a concurrent system is studied. The next section explains why the problem of achieving asynchronous interprocess communication may be viewed as one of implementing shared registers, and the following section describes algorithms for doing this. These two sections are informal, and may be read without having read the formalism of Part I. The concepts introduced in Section 4 are formally defined in Section 6, and formal correctness proofs of the algorithms of Section 5 are given in Section 7. These latter two sections assume knowledge of the material in Part I.

## 4 The Nature of Asynchronous Communication

All communication ultimately involves a communication medium whose state is changed by the sender and observed by the receiver. A sending processor changes the voltage on a wire and a receiving processor observes the voltage change; a speaker changes the vibrational state of the air and a listener senses this change.

There are two kinds of communication acts: *transient* and *persistent*. In a transient communication act, the medium’s state is changed only for the duration of the act, immediately afterwards reverting to its “normal” state. A message sent on an Ethernet modifies the transmission medium’s state only while the message is in transit; the altered state of the air lasts only while the speaker is talking. In a persistent communication act, the state change remains after the sender has finished its communication. Setting a voltage level on a wire, writing on a blackboard, and raising a flag on a flagpole are all examples of persistent communication.

Transient communication is possible only if the receiver is observing the communication medium while the sender is modifying it. This implies an *a priori* synchronization—the receiver must be waiting for the communication to take place. Communication between truly asynchronous processes must be persistent, the sender changing the state of the medium and the receiver able to sense that change at a later time.

At a low level, message passing is often considered to be a form of trans-

sient communication between asynchronous processes. However, a closer examination of asynchronous message passing reveals that it involves a persistent communication. Messages are placed in a buffer that is periodically tested by the receiver. Viewed at a low level, message passing is typically accomplished by putting a message in a buffer and setting an interrupt bit that is tested on every machine instruction. The receiving process actually consists of two asynchronous subprocesses: a *main* process that is usually thought of as the receiver, and an *input* process that continuously monitors the communication medium and transfers messages from the medium to the buffer. The input process is synchronized with the sender (it is a “slave” process) and communicates asynchronously with the main process, using the buffer as a medium for persistent communication.

The subject of this paper is asynchronous interprocess communication, so only persistent communication is considered. Moreover, attention is restricted to unidirectional communication, in which only a single process can modify the state of the medium. (With this restriction, two-way communication requires at least two separate communication media, one modified by each process.) However, multiple receivers will be considered. Also, only discrete systems, in which the medium has a finite number of distinguishable states, are considered. A receiver is assumed always to obtain one of these discrete values. The sender can therefore set the medium to one of a fixed number of persistent states, and the receiver(s) can observe the medium’s state.

This form of persistent communication is more commonly known as a shared register, where the sender and receiver are called the *writer* and *reader*, respectively, and the state of the communication medium is known as the *value* of the register. These terms are used in the rest of this paper, which therefore considers finite-valued registers with a single writer and one or more readers.

In assuming a single writer, the possibility of concurrent writes (to the same register) is ruled out. Since a reader only senses the value of the register, there is no reason why a read operation must interfere with another read or write operation. (While reads do interfere with other operations in some forms of memory, such as magnetic core, this interference is an idiosyncrasy of the particular technology rather than an inherent property of reading.) A read is therefore assumed not to affect any other read or any write. However, it is not clear what effect a concurrent write should have on a read.

In concurrent programming, one traditionally assumes that a writer has

exclusive access to shared data, making concurrent reading and writing impossible. This assumption is enforced either by requiring the programming language to provide the necessary exclusive access, or by implementing the exclusion with a “readers-writers” protocol [3]. Such an approach requires that a reader wait while a writer is accessing the register, and vice versa. Moreover, any method for achieving such exclusive access, whether implemented by the programmer or the compiler, requires a lower-level shared register. At some level, the problem of concurrent access to a shared register must be faced. It is this problem that is addressed by this paper; any approach that requires one process to wait for another is eschewed.

Asynchronous concurrent access to shared registers is usually considered only at the hardware level, so it is at this level that the methods developed here could have some direct application. However, concurrent access to shared data also occurs at higher levels of abstraction. One cannot allow any single process exclusive access to the entire Social Security system’s database. While algorithms for implementing a single register cannot be applied to such a database, I hope that insight obtained from studying these algorithms will eventually lead to new methods for higher-level data sharing. Nevertheless, when reading this paper, it is best to think of a register as a low-level component, probably implemented in hardware.

Hardware implementations of asynchronous communication often make assumptions about the relative speeds of the communicating processes. Such assumptions can lead to simplifications. For example, the problem of constructing an atomic register, discussed below, is shown to be easily solved by assuming that two successive reads of a register cannot be concurrent with a single write. If one knows how long a write can take, a delay can be added between successive reads to ensure that this assumption holds. No such assumptions are made here about process speeds. The results therefore apply even to communication between processes of vastly differing speeds.

Writes cannot overlap (be concurrent with) one another because there is only one writer, and overlapping reads are assumed not to affect one another, so the only case left to consider is a read overlapping one or more writes. Three possible assumptions about what can happen in this case are considered.

The weakest possibility is a *safe* register, in which it is assumed only that a read not concurrent with any write obtains the correct value—that is, the most recently written one. No assumption is made about the value obtained by a read that overlaps a write, except that it must obtain one of the possible values of the register. Thus, if a safe register may assume

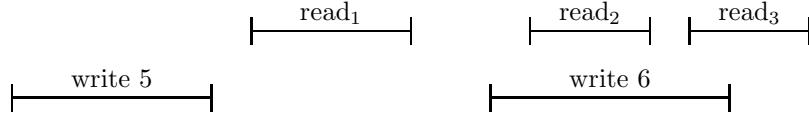


Figure 5: Two writes and three reads.

the values 1, 2, and 3, then any read must obtain one of these three values. A read that overlaps a write operation that changes the value from 1 to 2 could obtain any of these values, including 3.

The next stronger possibility is a *regular* register, which is safe (a read not concurrent with a write gets the correct value) and in which a read that overlaps a write obtains either the old or new value. For example, a read that overlaps a write that changes the value from 1 to 3 may obtain either 1 or 3, but not 2. More generally, a read that overlaps any series of writes obtains either the value before the first of the writes or one of the values being written.

The final possibility is an *atomic* register, which is safe and in which reads and writes behave as if they occur in some definite order. In other words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping. (The precise formal condition was developed in Section 2 of Part I.)

The difference between the three kinds of registers is illustrated by Figure 5, which shows five operations to a register that may assume the three values 5, 6, and 27. The duration of each operation is indicated by a line segment, where time runs from left to right. A write of the value 5 precedes all other operations, including a subsequent write of 6. There are three successive reads, denoted  $\text{read}_1$ ,  $\text{read}_2$ , and  $\text{read}_3$ .

For a safe register,  $\text{read}_1$  obtains the value 5, since a read that does not overlap a write must obtain the most recently written value. However, the other two reads, which overlap the second write, may obtain 5, 6, or 27.

With a regular register,  $\text{read}_1$  must again obtain the value 5, since a regular register is also safe. Each of the other two reads may obtain either a 5 or a 6, but not a 27. In particular,  $\text{read}_2$  could obtain a 6 and  $\text{read}_3$  a 5.

With an atomic register,  $\text{read}_1$  must also obtain the value 5 and the other two reads may obtain the following pairs of values:

| <u>read<sub>2</sub></u> | <u>read<sub>3</sub></u> |
|-------------------------|-------------------------|
| 5                       | 5                       |
| 5                       | 6                       |
| 6                       | 6                       |

For example, the pair of values 5,6 represents a situation in which the operations act as if the first read preceded the write of 6 and the second read followed it. However, unlike a regular register, an atomic register does not admit the possibility of read<sub>2</sub> obtaining the value 6 and read<sub>3</sub> obtaining 5. In general, if two successive reads overlap the same write, then a regular register allows the first read to obtain the new value and the second read the old value, while this is forbidden with an atomic register. In fact, Proposition 5 of Section 6 essentially states that a regular register is atomic if two successive reads that overlap the same write cannot obtain the new then the old value. Thus, a regular register is automatically an atomic one if two successive reads cannot overlap the same write.

These are the only three general classes of register that I have been able to think of. Each class merits study. Safeness<sup>5</sup> seems to be the weakest requirement that allows useful communication; I do not know how to achieve any form of interprocess synchronization with a weaker assumption. Regularity asserts that a read returns a “reasonable” value, and seems to be a natural requirement. Atomicity is the most common assumption made about shared registers, and is provided by current multiport computer memories.<sup>6</sup> At a lower level, such as interprocess communication within a single chip, only safe registers are provided; other classes of register must be implemented using safe ones.

Any method of implementing a single-writer register can be classified by three “coordinates” with the following values:

- *safe*, *regular*, or *atomic*, according to the strongest assumption that the register satisfies.
- *boolean* or *multivalued*, according to whether the method produces only boolean registers or registers with any desired number of values.

---

<sup>5</sup>The term “safeness” is used because “safety” already has a technical meaning for concurrent programs.

<sup>6</sup>However, the standard implementation of a multiport memory does not meet my requirements for an asynchronous register because, if two processes concurrently access a memory cell, one must wait for the other.

- *single-reader* or *multireader*, according to whether the method yields registers with only one reader or with any desired number of readers.

This produces twelve classes of implementations, partially ordered by “strength”—for example, a method that produces atomic, multivalued, multireader registers is stronger than one producing regular, multivalued, single-reader registers. This paper addresses the problem of implementing a register of one class using one or more registers of a weaker class.

The weakest class of register, and therefore the easiest to implement, is a safe, boolean, single-reader one. This seems to be the most natural kind of register to implement with current hardware technology, requiring only that the writer set a voltage level either high or low and that the reader test this level without disturbing it.<sup>7</sup> A series of constructions of stronger registers from weaker ones is presented that allows almost every class of register to be constructed starting from this weakest class. The one exception is that constructing an atomic, multireader register from any weaker one is still an open problem. Most of the constructions are simple; the difficult ones are Construction 4 that implements an  $m$ -reader, multivalued, regular register using  $m$ -reader, boolean, regular registers, and Construction 5 that implements a single-reader, multivalued, atomic register using single-reader, multivalued, regular registers.

## 5 The Constructions

In this section, the algorithms for constructing different classes of registers are described and informally justified. Rigorous correctness proofs are postponed until Section 7.

The algorithms are described by indicating how a write and a read are performed. For most of them, the initial state is not indicated—it is the one that would result from writing the initial value starting from any arbitrary state.

The first construction implements a multireader safe or regular register from single-reader ones. It uses the obvious method of having the writer maintain a separate copy of the register for each reader. The **for all** statement denotes that its body is executed once for each of the indicated values of  $i$ ; these separate executions can be done in any order or concurrently.

---

<sup>7</sup>This is only safe and not regular if, for example, setting a level high when it is already high can cause a perturbation of the level.

**Construction 1** Let  $v_1, \dots, v_m$  be single-reader,  $n$ -valued registers, where each  $v_i$  can be written by the same writer and read by process  $i$ , and construct a single  $n$ -valued register  $v$  in which the operation  $v := \mu$  is performed as follows:

**for all**  $i$  in  $\{1, \dots, m\}$  **do**  $v_i := \mu$  **od**

and process  $i$  reads  $v$  by reading the value of  $v_i$ . If the  $v_i$  are safe or regular registers, then  $v$  is a safe or regular register, respectively.

The proof of correctness for this construction runs as follows. Any read by process  $i$  that does not overlap a write of  $v$  does not overlap a write of  $v_i$ . If  $v_i$  is safe, then this read gets the correct value, which shows that  $v$  is safe. If a read of  $v_i$  by process  $i$  overlaps a write of  $v_i$ , then it overlaps the write of the same value to  $v$ . This implies that if  $v_i$  is regular, then  $v$  is also regular.

Construction 1 does not make  $v$  an atomic register even if the  $v_i$  are atomic. If reads by two different processes  $i$  and  $j$  both overlap the same write, it is possible for  $i$  to get the new value and  $j$  the old value even though the read by  $i$  precedes the read by  $j$ —a possibility not allowed by an atomic register.

The next construction is also trivial; it implements an  $n$ -bit safe register from  $n$  single-bit ones.

**Construction 2** Let  $v_1, \dots, v_n$  be boolean  $m$ -reader registers, each written by the same writer and read by the same set of readers. Let  $v$  be the  $2^n$ -valued,  $m$ -reader register in which the number with binary representation  $\mu_1 \dots \mu_n$  is written by

**for all**  $i$  in  $\{1, \dots, m\}$  **do**  $v_i := \mu_i$  **od**

and in which the value is read by reading all the  $v_i$ . If each  $v_i$  is safe, then  $v$  is safe.

This construction yields a safe register because, by definition, a read does not overlap a write of  $v$  only if it does not overlap a write of any of the  $v_i$ , in which case it obtains the correct values. The register  $v$  is not regular even if the  $v_i$  are. A read can return any value if it overlaps a write that changes the register's value from  $0 \dots 0$  to  $1 \dots 1$ .

The next construction shows that it is trivial to implement a boolean regular register from a safe boolean register. In a safe register, a read that

overlaps a write may get any value, while in a regular register it must get either the old or new value. However, a read of a safe boolean register must obtain either *true* or *false* on any read, so it must return either the old or new value if it overlaps a write that changes the value. A boolean safe register can fail to be regular only if a read that overlaps a write that does not change the value returns the other value—for example, writing the value *true* when the current value equals *true* could cause an overlapping read to obtain the value *false*. To prevent this possibility, one simply does not perform a write that does not change the value.

**Construction 3** *Let  $v$  be an  $m$ -reader boolean register, and let  $x$  be a variable internal to the writer (not a shared register) initially equal to the initial value of  $v$ . Define  $v^*$  to be the  $m$ -reader boolean register in which the write operation  $v^* := \mu$  is performed as follows:*

```
if $x \neq \mu$ then $v := \mu$;
 $x := \mu$ fi
```

*and a read of  $v^*$  is performed by reading  $v$ . If  $v$  is safe then  $v^*$  is regular.*

There are two known algorithms for implementing a multivalued regular register from boolean ones. The simpler one is given as Construction 4; the second one is described later. Construction 4 employs a unary encoding, in which the value  $\mu$  is denoted by zeros in bits 0 through  $\mu - 1$  and a one in bit  $\mu$ . A reader reads the bits from left to right (0 to  $n$ ) until it finds a one. To write the value  $\mu$ , the writer first sets  $v_\mu$  to one and then sets bits  $\mu - 1$  through 1 to zero, writing from right to left. (While this algorithm has never before been published, the idea of implementing shared data by reading and writing its components in different directions was also used in [4].<sup>8</sup>)

**Construction 4** *Let  $v_1, \dots, v_n$  be boolean,  $m$ -reader registers, and let  $v$  be the  $n$ -valued,  $m$ -reader register in which the operation  $v := \mu$  is performed by*

```
 $v_\mu := 1$;
for $i := \mu - 1$ step -1 until 1 do $v_i := 0$ od
```

---

<sup>8</sup>Although the algorithms in [4] require only that the registers be regular, the assumption of atomicity was added because the editor felt that nonatomicity at the level of individual bits was too radical a concept to appear in *Communications of the ACM*.

and a read is performed by:

```

 $\mu := 1;$
while $v_\mu = 0$ do $\mu := \mu + 1$ od;
return μ

```

If each  $v_i$  is regular, then  $v$  is regular.

The correctness of this algorithm is not at all obvious. Indeed, it is not even obvious that the **while** loop in the read operation does not “fall off the end” and try to read the nonexistent register  $v_{n+1}$ . This can’t happen because, whenever the writer writes a zero, there is a one to the right of it. (Since an initial value is assumed to have been written, some  $v_i$  initially equals one.) As an exercise, the reader of this paper can convince himself that, whenever a reading process sees a one, it was written by either a concurrent write or by the most recent preceding one, so  $v$  is regular. The formal proof is given in Section 7.

The value of  $v_n$  is only set to one, never to zero. It can therefore be eliminated; the writer simply never writes it and the reader assumes its value is one instead of reading it.

Even if all the  $v_i$  are atomic, Construction 4 does not produce an atomic register. To see this, suppose that the register initially has the value 3, so  $v_1 = v_2 = 0$  and  $v_3 = 1$ , the writer first writes the value 1 then the value 2, and there are two successive read operations. This can produce the following sequence of actions:

- the first read finds  $v_1 = 0$
- the first write sets  $v_1 := 1$
- the second write sets  $v_2 := 1$
- the first read finds  $v_2 = 1$  and returns the value 2
- the second read finds  $v_1 = 1$  and returns the value 1.

In this scenario, the first read obtains a newer value (the one written by the second write) than the second read (which obtains the one written by the first write), even though it precedes the second read. This shows that the register is not atomic.

Construction 4 uses  $n - 1$  boolean regular registers to make an  $n$ -valued one, so it is practical only for small values of  $n$ . One would like an algorithm that requires  $O(\log n)$  boolean registers to construct an  $n$ -valued

register. The second method for constructing a regular multivalued register uses an algorithm of Peterson [14] that implements an  $m$ -reader,  $n$ -valued, atomic register with  $m + 2$  safe,  $m$ -reader,  $n$ -valued registers;  $2m$  atomic, boolean, one-reader registers; and two atomic, boolean  $m$ -reader registers. However, there is no known algorithm for constructing the atomic,  $m$ -reader registers required by Peterson's algorithm from simpler ones. Nevertheless, we can apply his algorithm to construct an  $n$ -valued, single-reader, atomic register using three safe, single-reader,  $n$ -valued registers and four single-reader, atomic, boolean registers. The safe registers can be implemented with Construction 2, and the atomic boolean registers can be implemented with Construction 5 below. Since an atomic register is regular, Construction 1 can then be used to make an  $m$ -reader,  $n$ -valued, regular register from  $O(3m \log n)$  single-reader, boolean, regular registers.

Before giving the algorithm for constructing a two-reader atomic register, a result is proved that indicates why no trivial algorithm will work. It asserts that there can be no algorithm in which the writer only writes and the reader only reads; any algorithm must involve two-way communication between the reader and the writer.

**Theorem:** *There exists no algorithm to implement an atomic register using a finite number of regular registers that can be written only by the writer (of the atomic register).*

*Proof:* We assume such an algorithm and derive a contradiction. Any algorithm that uses multiple registers can be replaced by one in which these registers are combined into a single large register. A read in the original algorithm is replaced by one that reads all the combined register and ignores the other components; a write in the original algorithm is replaced by one that changes only the desired component of the combined register. (This is possible because there is only a single writer.) Therefore, without loss of generality, we can assume that there is only a single regular register  $v$  written by the writer and read by the reader.

Let  $v^*$  denote the atomic register that is being implemented. Since the algorithm must work if the writer never stops writing, we may suppose that the writer performs an infinite number of writes that change the value of  $v^*$ . There must be some pair of values assumed by  $v^*$ , call them 0 and 1, such that there are an infinite number of writes that change  $v^*$ 's value from 0 to 1. Since  $v$  can assume only a finite number of values (the hypothesis states that the original algorithm has only a finite number of registers, and

all registers are taken to have only a finite number of possible values), there must exist values  $v_0, \dots, v_n$  of  $v$  such that: (i)  $v_0$  is the final value of  $v$  after each one of an infinite number of writes of 0 to  $v^*$ , (ii)  $v_n$  is the final value of  $v$  after each one of an infinite number of writes of 1 to  $v^*$ , and (iii) for each  $i < n$ , the value of  $v$  is changed from  $v_i$  to  $v_{i+1}$  during infinitely many writes that change the value of  $v^*$  from 0 to 1.<sup>9</sup>

A read of  $v^*$  may involve several reads of  $v$ . However, in our quest for a contradiction, we may restrict our attention to scenarios in which each of those reads of  $v$  obtains the same value, so we may assume that each read of  $v^*$  reads  $v$  only once. Since  $v$  assumes each value  $v_i$  infinitely often, it must be possible for a sequence of  $n+1$  consecutive reads of  $v$  to obtain the values  $v_n, v_{n-1}, \dots, v_0$ .

The read that finds  $v$  equal to  $v_i$  and the subsequent read that finds  $v$  equal to  $v_{i-1}$  could both have overlapped the same write of  $v$ , which could have been a write that occurred in the process of changing  $v^*$ 's value from 0 to 1. Therefore, if the read of  $v^*$  that finds  $v$  equal to  $v_i$  returns the value 1, then the subsequent read that finds  $v$  equal to  $v_{i-1}$  must also return the value 1, since both reads could be overlapping the same write and, in that case, two successive reads of an atomic register cannot return first the new value, then the old one.

The first read, which finds  $v$  equal to  $v_n$ , must return the value 1, since it could have occurred after the completion of a write of 1. By induction, this implies that the last read, which found  $v$  equal to  $v_0$ , must return the value 1. However, this read could have occurred after a write of 0 and before any subsequent write, so returning the value 1 would violate the assumption that the register  $v^*$  is safe. (An atomic register is *a fortiori* safe.) This is the required contradiction. ■

This theorem could be expressed and proved using the formalism of Part I and the definitions of the next section, but doing so would lead to no new insight. The formalization of this theorem is therefore left as an exercise for the reader who wishes to gain practice in using the formalism.

The theorem is false if no bound is placed on the number of values a register can hold. Given a regular register  $v$  that can assume an unbounded

---

<sup>9</sup>If we assume that the writer has only a finite number of internal states, then we can conclude that the precise sequence of values  $v_0, \dots, v_n$  is written infinitely many times when changing the value of  $v^*$  from 0 to 1. However, with an infinite number of internal states, it is possible for the writer never to perform the same sequence of writes to  $v$  twice.

number of values, an atomic register  $v^*$  is implemented as follows. The writer sets  $v$  equal to a pair consisting of the value of  $v^*$  and a sequential version number. The reader reads  $v$  and compares the version number with the previous one it read. If the new version number is higher, then it uses the value it just read; if the new version number is lower, then it forgets the value and version number it just read and uses the previously read value. The correctness of this algorithm follows easily from Proposition 5 of Section 6. By assuming that registers hold only a bounded set of values, such algorithms are disallowed.

Finally, we come to the algorithm for constructing a single-reader, multi-valued, atomic register from regular ones. Let  $v^*$  denote the atomic register being implemented, and let the writer set this register by writing into a shared regular register  $v$ . Suppose that some value  $\mu$  of  $v^*$  is represented by letting  $v$  equal  $v_0$ , and that to change the value of  $v^*$  to another value  $\nu$ , the writer successively sets  $v$  to the values  $v_1, v_2, \dots, v_n$ , where  $v = v_n$  represents  $v^* = \nu$ . The proof of the above theorem rested upon showing that the reader is in a quandary if  $n$  successive reads return the values  $v_n, v_{n-1}, \dots, v_0$ . If the  $i^{\text{th}}$  read returns  $\nu$  then the  $i + 1^{\text{st}}$  read must also return  $\nu$  because both reads could have overlapped the same write of  $v$ , in which case returning  $\mu$  would result in the later read returning the earlier value. The first read must return the value  $\nu$ , so the last read, which ought to return  $\mu$ , the value of  $v^*$  denoted by  $v = v_0$ , is forced to return  $\nu$ .

The way out of this problem is to encode, as part of  $v$ 's value, a boolean quantity called a *color*. Each value of  $v^*$  is represented by two different values of  $v$ —one of each color. Every time the reader reads  $v$ , it sets a boolean register  $c$  to the color of the value it just read. When the writer wants to write a new value of  $v^*$ , it first reads  $c$  and then makes the series of values  $v_1, \dots, v_n$  it writes to  $v$  have the opposite color to  $c$ . (Thus, the reader tries to keep  $c$  equal to the color of  $v$ , and the writer tries to keep the color of  $v$  different from  $c$ .) It can be shown that if  $n \geq 4$ , so at least three intermediate values are written when changing the value of  $v^*$ , then successive reads cannot obtain the sequence  $v_n, \dots, v_0$ . This enables one to devise an algorithm in which the writer changes the value of the register from  $\mu$  to  $\nu$  by first writing a series of intermediate values  $(\mu, \nu, 1, \kappa), (\mu, \nu, 2, \kappa), (\mu, \nu, 3, \kappa)$ , and then writing  $(\nu, \kappa)$ , where  $\kappa$  is the color. However, one can do better, and an algorithm is developed below that uses only two intermediate values.

When  $n = 3$ , so the writer writes the sequence  $v_1, v_2, v_3$ , with  $v_3$  representing the new value  $\nu$ , it is possible for three successive reads  $R_3, R_2,$

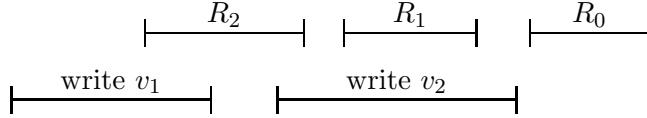


Figure 6: Reads  $R_2$  and  $R_1$  overlapping the write of  $v_2$ .

$R_1$  to obtain the values  $v_3$ ,  $v_2$ , and  $v_1$ , respectively. However, it will be shown that this can happen only if the two reads  $R_2$  and  $R_1$  both overlap a single write of  $v_2$ . As indicated by Figure 6, this implies that a fourth read  $R_0$  cannot overlap the write of the value  $v_1$  that was obtained by  $R_1$ . Therefore, if the fourth read  $R_0$  obtains  $v_0$ , the reader can return the value  $\mu$  represented by  $v_0$  with no fear of the pair  $R_1$ ,  $R_0$  returning a forbidden “new-old” sequence.

The following construction implements an atomic register  $v^*$  using two regular registers  $v$  (written by the writer) and  $c$  (written by the reader). For clarity, it is presented in a form in which  $v$  can assume more values than are necessary; the number of different values that  $v$  really needs is discussed afterwards. To change the value of  $v^*$  from  $\mu$  to  $\nu$ , the writer first sets  $nc$  to be different from  $c$ , then writes the following sequence of values to  $v$ :  $(\mu, \nu, 1, nc)$ ,  $(\mu, \nu, 2, nc)$ ,  $(\mu, \nu, 3, nc)$ . Thus,  $v = (\mu, \nu, 3, \kappa)$  denotes  $v^* = \nu$  for any values of  $\mu$  and  $\kappa$ .

The reader reads  $v$  and sets  $c$  equal to its color, but what value of  $v^*$  does it return? Suppose the reader obtains the value  $(\mu, \nu, i, \kappa)$  when reading  $v$ . If  $i = 3$ , then to guarantee safeness, the reader must return  $\nu$ . If  $i < 3$ , then regularity requires only that the read return either  $\mu$  or  $\nu$ . The basic idea is for the reader to return  $\mu$  except when this might allow the possibility that two successive reads overlapping the same write return first the new then the old value. For example, this is the case if the preceding read had obtained the value  $(\mu, \nu, i + 1, \kappa)$  and returned the value  $\nu$ . To simplify the algorithm, the reader bases its decision of which value to return only upon the values of  $i$  and  $\kappa$  obtained by this and the preceding read, not upon the values of  $\mu$  and  $\nu$  obtained by the preceding read.

The following notation is used in describing the algorithm: for  $\xi = (\mu, \nu, i, \kappa)$ , let  $old(\xi) = \mu$ ,  $new(\xi) = \nu$ ,  $num(\xi) = i$ , and  $col(\xi) = \kappa$ . In the algorithm, the variable  $v$  is written by the writer and read by both the reader and the writer. A two-reader register is not needed, since the writer can maintain a local variable containing the value that it last wrote into  $v$ . (This is just Construction 1 with  $m = 2$  and the writer being the

second reader.) Such a local variable would complicate the description, so it is omitted. The variables  $nc$ ,  $\mu$ ,  $rv$ ,  $rv'$ , and  $nuret$  are local (not shared registers);  $nuret$  is true if the reader returned the “ $\nu$  value” on the preceding read. The proof of correctness of this construction is given in Section 7.

**Construction 5** *Let  $w$  and  $r$  be processes, let  $\mathcal{V}^*$  be a finite set, let  $v$  be a regular register with values in  $\mathcal{V}^* \times \mathcal{V}^* \times \{1, 2, 3\} \times \{\text{true}, \text{false}\}$  that can be written by  $w$  and read by  $r$ , with  $\text{num}(v)$  initially equal to 3, and let  $c$  be a regular boolean register that can be written by  $r$  and read by  $w$ . Define the register  $v^*$  with values in  $\mathcal{V}^*$ , written by  $w$  and read by  $r$ , as follows. The write  $v^* := \nu$  is performed by*

```

 $nc := \neg c;$
 $\mu := \text{old}(v);$
for $i := 1$ until 3 do $v := (\mu, \nu, i, nc)$

```

*and the read operation is performed by the following algorithm, where  $nuret$  is initially false:*

```

 $rv' := rv;$
 $rv := v;$
 $c := \text{col}(rv);$
if $\text{num}(rv) = 3$
 then $nuret := \text{true};$
 return $\text{new}(rv)$
 else if $nuret \wedge \text{col}(rv) = \text{col}(rv') \wedge \text{num}(rv) \geq \text{num}(rv') - 1$
 then return $\text{new}(rv)$
 else $nuret := \text{false};$
 return $\text{old}(rv)$
fi fi

```

*Then  $v^*$  is an atomic register.*

If a read  $R_1^*$  of  $v^*$  obtains the value  $(\mu, \nu, 1, \kappa)$  for  $v$  and returns  $\nu$  as the value of  $v^*$ , then there must have been two previous reads  $R_3^*$  and  $R_2^*$  that obtained the values  $(\dots, 3, \kappa)$  and  $(\dots, 2, \kappa)$ , respectively, for  $v$  such that any reads coming between  $R_3^*$  and  $R_1^*$  obtained a value  $(\dots, \kappa)$ . It will be shown in the correctness proof of the construction that this can happen only if  $R_2^*$  obtained the value  $(\mu, \nu, 2, \kappa)$ . This means that the read  $R_1^*$  can simply return the same value returned by  $R_2^*$ . Hence, if the reader remembers the

last value returned by a read that found  $num(rv) = 2$ , then the  $\nu$  component is redundant in values of  $v$  of the form  $(\mu, \nu, 1, \kappa)$ .

When  $num(rv) = 3$ , the reader always returns  $new(rv)$ . Hence, the  $\mu$  component is redundant in values of  $v$  of the form  $(\mu, \nu, 3, \kappa)$ . Since the writer can simply do nothing if the value it is writing is the same as the current value, there is no need for  $v$  to assume values in which  $\mu = \nu$ .

From these observations, it follows that  $v$  need assume only values of the following forms, with  $\mu \neq \nu$ :  $(\mu, 1, \kappa)$ ,  $(\mu, \nu, 2, \kappa)$ , and  $(\nu, 3, \kappa)$ . If there are  $n$  possible values for  $\mu$  and  $\nu$ , then there are  $2n(n+2)$  such values. Therefore, Construction 5 can be modified to implement an  $n$ -valued atomic register  $v^*$  with a  $2n(n+2)$ -valued regular register  $v$  written by the writer and read by the reader and a boolean regular register  $c$  written by the reader and read by the writer.

## 6 Register Axioms

The formalism described in Part I applies to any system execution. For system executions containing reads and writes to registers, the general axioms A1–A5 of Part I must be augmented by axioms for these operation executions. They include axioms that provide the formal statements of the properties of safe, regular, and atomic registers.

Axioms A1–A5 do not require that there be any precedence relations among operation executions. However, some precedence relations must be assumed among operations to the same register. Implicit in our assumption that a register has only a single writer is the assumption that all the writes to a register are totally ordered. We let  $V^{[1]}, V^{[2]}, \dots$  denote the sequence of write operations to the register  $v$ , where  $V^{[1]} \longrightarrow V^{[2]} \longrightarrow \dots$  and let  $v^{[i]}$  denote the value written by  $V^{[i]}$ . (There may be a finite or infinite number of write operations  $V^{[i]}$ .)

A register  $v$  is assumed to have some initial value  $v^{[0]}$ . It is convenient to assume that this value is written by a write  $V^{[0]}$  that precedes ( $\longrightarrow$ ) all other reads and writes of  $v$ . Eliminating this assumption changes none of the results, but it complicates the reasoning because a read that precedes all writes has to be treated as a separate case. These assumptions are expressed formally by the following axiom.

- B0. The set of write operation executions to a register  $v$  consists of the (finite or infinite) set  $\{V^{[0]}, V^{[1]}, \dots\}$  where  $V^{[0]} \longrightarrow V^{[1]} \longrightarrow \dots$  and,

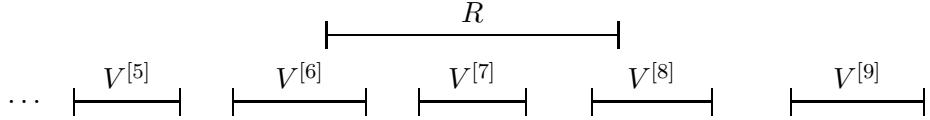


Figure 7: A read that sees  $v^{[5,8]}$ .

for any read  $R$  of  $v$ ,  $V^{[0]} \longrightarrow R$ . The value written by  $V^{[i]}$  is denoted  $v^{[i]}$ .

Communication implies causal connection; for processes to communicate through operations to a register, there must be some causality ( $\dashrightarrow$ ) relations between reads and writes of the register. The following axiom is therefore assumed; the reader is referred to [6] (where it is labeled C3) for its justification.

- B1. For any read  $R$  and write  $W$  to the same register,  $R \dashrightarrow W$  or  $W \dashrightarrow R$  (or both).

Note that B1 holds for any system execution that has a global-time model because, for any operation executions  $A$  and  $B$  in such a system execution, either  $A \longrightarrow B$  or  $B \dashrightarrow A$ .

Each register has a finite set of possible values—for example, a boolean-valued register has the possible values *true* and *false*. A read is assumed to obtain one of these values, whether or not it overlaps a write.

- B2. A read of a register obtains one of the (finite collection of) values that may be written in the register.

Thus, a read of a boolean register cannot obtain a nonsense value like “*trlse*”. Axiom B2 does not assert that the value obtained by a read was ever actually written in the register, so it does not imply safeness.

Let  $R$  be a read of register  $v$ , and let

$$\begin{aligned} I_R &\stackrel{\text{def}}{=} \{V^{[k]} : R \not\longrightarrow V^{[k]}\} \\ J_R &\stackrel{\text{def}}{=} \{V^{[k]} : V^{[k]} \dashrightarrow R\} \end{aligned}$$

In the example of Figure 7,  $I_R = \{V^{[0]}, \dots, V^{[5]}\}$  and  $J_R = \{V^{[0]}, \dots, V^{[8]}\}$ . As this example shows, in system executions with a global-time model,  $I_R$  is the set of writes that precede ( $\longrightarrow$ )  $R$  and the writes in  $J_R$  are the ones that could causally affect  $R$ . The difference  $J_R - I_R$  of these two sets is the

set of writes that are concurrent with (overlap)  $R$ . If we think of the register as containing “traces” of both the old and new values during a write, then a read  $R$  can see traces of the values written by writes in  $J_R - I_R$  and by the last write in  $I_R$ . In Figure 7,  $R$  can see traces of the values  $v^{[5]}$  through  $v^{[8]}$ . (The value  $v^{[5]}$  is present during the write  $V^{[6]}$ , which is overlapped by  $R$ .) All traces of earlier writes vanish with the completion of the last write in  $I_R$ , and  $R$  sees no value written after the last write in  $J_R$ . This suggests the following formal definition, where “sees  $v^{[i,j]}$ ” is an abbreviation for “sees traces of  $v^{[i]}$  through  $v^{[j]}$ ”.

**Definition 9** *A read  $R$  of register  $v$  is said to see  $v^{[i,j]}$  where:*

$$\begin{aligned} i &\stackrel{\text{def}}{=} \max\{k : R \dashv \rightarrow V^{[k]}\} \\ j &\stackrel{\text{def}}{=} \max\{k : V^{[k]} \dashv \rightarrow R\} \end{aligned}$$

The informal discussion that led to this definition was based upon a global-time model. When the existence of a global-time model is not assumed,  $I_R$  not only contains all the writes that precede  $R$ , but it may contain later writes as well. The set  $J_R$  consists of all writes that could causally affect  $R$ .

For Definition 9 to make sense, it is necessary that the sets whose maxima are taken—or, equivalently, the sets  $I_R$  and  $J_R$ —be finite and nonempty. They are nonempty because, by A2 and the assumption that  $V^{[0]}$  precedes all reads, both  $I_R$  and  $J_R$  contain  $V^{[0]}$ ; and Axioms A5 and A2 imply that they are finite. Furthermore, B1 implies that  $I_R \subseteq J_R$ , so  $i \leq j$ .

The formal definitions of safe, regular, and live registers can now be given. A safe register has been informally defined to be one that obtains the correct value if it is not concurrent with any write. A read that is not concurrent with a write is one that sees traces of only a single write, which leads to the following definition:

B3. (*safe*) A read that sees  $v^{[i,i]}$  obtains the value  $v^{[i]}$ .

A regular register is one for which a read obtains a value that it “could have” seen—that is, a value it has seen a trace of.

B4. (*regular*) A read that sees  $v^{[i,j]}$  obtains a value  $v^{[k]}$  for some  $k$  with  $i \leq k \leq j$ .

An atomic register satisfies the additional requirement that a read is never concurrent with any write.

B5. (*atomic*) If a read sees  $v^{[i,j]}$  then  $i = j$ .

A safe register satisfies B0–B3, a regular register satisfies B0–B4 (note that B4 implies B3), and an atomic register satisfies B0–B5.

Observe that in B3–B5, the conditions placed upon the value obtained by a read  $R$  of register  $v$  depend only upon precedence relations between  $R$  and writes of  $v$ . No other operation executions affect  $R$ . In particular, a read is not influenced by other reads.

The following two propositions state some useful properties that are simple consequences of Definition 9. In Proposition 3, the notation is introduced that  $v^{[i,j]}$  denotes a read that sees the value  $v^{[i,j]}$ , so part (a) is an abbreviation for: “If  $R$  is a read that sees  $v^{[i,j]}$  and  $R \longrightarrow V^{[k]}$ , then  $\dots$ ” (Recall that  $V^{[k]}$  is the  $k^{\text{th}}$  write of  $v$ .)

**Proposition 3** (a) *If  $v^{[i,j]} \longrightarrow V^{[k]}$  then  $j < k$ .*

(b) *If  $V^{[k]} \longrightarrow v^{[i,j]}$  then  $k \leq i$ .*

(c) *If  $v^{[i,j]} \longrightarrow v^{[i',j']}$  then  $j \leq i' + 1$ .*

*Proof:* Parts (a) and (b) are immediate consequences of Definition 9. To prove part (c), observe first that Definition 9 also implies that  $V^{[j]} \dashrightarrow v^{[i,j]}$ . Part (c) is immediate if  $j = 0$ . If  $j > 0$ , then  $V^{[j-1]} \longrightarrow V^{[j]}$ . Combining these two relations with the hypothesis gives

$$V^{[j-1]} \longrightarrow V^{[j]} \dashrightarrow v^{[i,j]} \longrightarrow v^{[i',j']}$$

Axiom A4 implies that  $V^{[j-1]} \longrightarrow v^{[i',j']}$ , which, by A2, implies  $v^{[i',j']} \dashrightarrow V^{[j-1]}$ . Definition 9 then implies that  $j - 1 \leq i'$ . ■

**Proposition 4** *If  $R$  is a read that sees  $v^{[i,j]}$ , then*

(a)  *$k \leq j$  if and only if  $V^{[k]} \dashrightarrow R$ .*

(b)  *$i \leq k$  if and only if  $R \dashrightarrow V^{[k+1]}$ .*

*Proof:* To prove part (a), observe that it follows immediately from Definition 9 that  $V^{[k]} \dashrightarrow R$  implies  $k \leq j$ . To prove the converse, assume  $k \leq j$ . Since  $V^{[j]} \dashrightarrow R$ , the desired conclusion,  $V^{[k]} \dashrightarrow R$ , is immediate if  $k = j$ . If  $k < j$ , then  $V^{[k]} \longrightarrow V^{[j]}$ , and the result follows from A3.

For part (b), Definition 9 implies that if  $i < k'$  then  $R \dashrightarrow V^{[k']}$ . Letting  $k' = k + 1$ , this shows that if  $i \leq k$  then  $R \dashrightarrow V^{[k+1]}$ . Conversely, suppose  $R \dashrightarrow V^{[k+1]}$ . By Definition 9, this implies  $k + 1 \neq i$ . If  $k + 1 < i$ , then  $V^{[k+1]} \rightarrow V^{[i]}$ , so A3 would imply  $R \dashrightarrow V^{[i]}$ , contrary to Definition 9. Hence, we must have  $i < k + 1$ , so  $i \leq k$ , completing the proof of part (b).  $\blacksquare$

Atomicity is usually taken to mean that all reads and writes are totally ordered in time. With B5, atomicity is defined by the requirement that each individual read is totally ordered with respect to the writes, but it leaves the possibility that two reads may overlap. It can be shown that, given a system execution for an atomic register, the partial ordering  $\rightarrow$  can be completed to a total ordering of reads and writes without violating conditions B1–B5. Thus, a system containing an atomic register trivially implements one in which all reads and writes are sequentially ordered. (Recall the definition of a trivial implementation in Part I.)

The following proposition is used in the formal correctness proof of Construction 5.

**Proposition 5** *Let  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  be a system execution containing reads and writes to a regular register  $v$ . If there exists an integer-valued function  $\phi$  on the set of reads such that:*

1. *If  $R$  sees  $v^{[i,j]}$ , then  $i \leq \phi(R) \leq j$ .*
2. *A read  $R$  returns the value  $v^{[\phi(R)]}$ .*
3. *If  $R \rightarrow R'$  then  $\phi(R) \leq \phi(R')$ .*

*then  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  trivially implements a system execution in which  $v$  is an atomic register.*

*Proof:* Proposition 2 of Part I, with the set of reads and writes of  $v$  substituted for  $\mathcal{S}$  and with the set of all other operations in  $\mathcal{S}$  substituted for  $\mathcal{T}$ , shows that it suffices to prove the proposition under the assumption that  $\mathcal{S}$  consists entirely of the reads and writes of  $v$ .

Let  $\xrightarrow{1}$  be the relation on  $\mathcal{S}$  that is the same as  $\rightarrow$  except between reads and writes of  $v$ , and, for any read  $R$  and write  $V^{[k]}$  of  $v$ :  $V^{[k]} \xrightarrow{1} R$  if  $k \leq \phi(R)$ , and  $R \xrightarrow{1} V^{[k]}$  if  $k > \phi(R)$ . Let  $R$  be a read that sees  $v^{[i,j]}$ . If  $V^{[k]} \rightarrow R$ , then part (b) of Proposition 3 implies that  $k \leq i$ , so, by property 1 of  $\phi$ ,  $k \leq \phi(R)$ . By definition of  $\xrightarrow{1}$ , this implies  $V^{[k]} \xrightarrow{1} R$ .

Similarly, part (a) of Proposition 3 implies that if  $R \rightarrow V^{[k]}$  then  $R \xrightarrow{1} V^{[k]}$ . Hence,  $\xrightarrow{1}$  is an extension of  $\rightarrow$ .

By B0, the relation  $\xrightarrow{1}$  is a total ordering on writes, and by definition it totally orders each read with respect to the writes. The next step is to extend  $\xrightarrow{1}$  to a total ordering on  $\mathcal{S}$ , which requires extending it to a total ordering on the set of reads. The restriction of  $\xrightarrow{1}$  to the set of reads is just  $\rightarrow$ , which is an irreflexive partial ordering. By property 3 of  $\phi$ , we can therefore complete  $\xrightarrow{1}$  to a total ordering  $\xrightarrow{2}$  of the reads, such that if  $\phi(R) < \phi(R')$  then  $R \xrightarrow{2} R'$ .

Let  $\xrightarrow{3}$  be the union of  $\xrightarrow{1}$  and  $\xrightarrow{2}$ . It is clear that for any read and/or write operation executions  $A$  and  $B$ , either  $A \xrightarrow{3} B$  or  $B \xrightarrow{3} A$ . To show that  $\xrightarrow{3}$  is a total ordering—meaning that it is a complete partial ordering, where a partial ordering is transitively closed and irreflexive—it is necessary to show that it is acyclic. Since the restriction of  $\xrightarrow{1}$  to the writes is a total ordering and  $\xrightarrow{2}$  is a total ordering on the set of reads that extends  $\xrightarrow{1}$ , any cycle of  $\xrightarrow{3}$  must be of the form

$$W_1 \xrightarrow{1} R_1 \xrightarrow{2} \dots \xrightarrow{2} R_n \xrightarrow{1} W_2 \xrightarrow{1} R_{n+1} \xrightarrow{2} \dots \xrightarrow{1} W_1$$

where the  $W_i$  are writes and the  $R_j$  are reads. But such a cycle is impossible because of the following three observations, where  $R$  is any read, the first two coming from the definition of  $\xrightarrow{1}$  and the second from the definition of  $\xrightarrow{2}$ :

- (a)  $V^{[k]} \xrightarrow{1} R$  implies  $k \leq \phi(R)$
- (b)  $R \xrightarrow{1} V^{[k]}$  implies  $\phi(R) < k$
- (c)  $R \xrightarrow{2} R'$  implies  $\phi(R) \leq \phi(R')$

Thus,  $\xrightarrow{3}$  is a total ordering of  $\mathcal{S}$  that extends  $\rightarrow$ . Letting  $\xrightarrow{3}$  equal  $\xrightarrow{3}$  then makes  $\langle \mathcal{S}, \xrightarrow{3}, \xrightarrow{3} \rangle$  a system execution. (Axioms A1–A4 follow easily from the fact that  $\xrightarrow{3}$  is a total ordering, and A5 follows from the fact that  $\xrightarrow{3}$  extends  $\rightarrow$ , for which A5 holds.) Thus,  $\langle \mathcal{S}, \rightarrow, \xrightarrow{3} \rangle$  trivially implements  $\langle \mathcal{S}, \xrightarrow{3}, \xrightarrow{3} \rangle$ . To complete the proof of the proposition, it suffices to show that  $\langle \mathcal{S}, \xrightarrow{3}, \xrightarrow{3} \rangle$  satisfies B0–B5.

Property B0 is trivial, since it holds for  $\rightarrow$  and  $\xrightarrow{3}$  is the same as  $\rightarrow$  on the set of writes. Property B1 is also trivial, since  $\xrightarrow{3}$  is a total

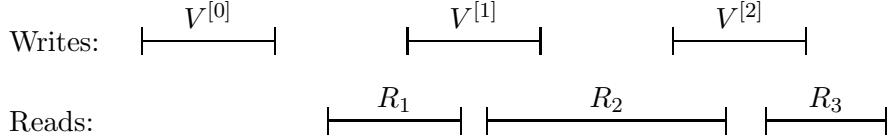


Figure 8: An interesting collection of reads and writes.

ordering. Property B2 follows from the corresponding property for  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ . To prove the remaining properties, observe that the definition of  $\xrightarrow{1}$  implies that, in the system execution  $\langle \mathcal{S}, \xrightarrow{3}, \dashrightarrow \rangle$ , any read  $R$  sees  $v^{[\phi(R), \phi(R)]}$ . Properties B3–B5 then follow immediately from the assumption that a read  $R$  obtains the value  $v^{[\phi(R)]}$ . ■

It was observed above that a regular register can fail to be atomic because two successive reads that overlap the same write could return the new then the old value. Intuitively, Proposition 5 shows that this is the only way a regular register can fail to be atomic. To see this, observe that a function  $\phi$  satisfying properties 1 and 2 of the proposition exists if and only if  $v$  is regular. The third property states that two consecutive reads do not obtain out-of-order values.

The exact wording of the proposition is important. One might be tempted to replace the hypothesis with the weaker requirement that  $v$  be regular and the following hold:

- 3' If  $v^{[i,j]} \rightarrow v^{[i',j']}$  then there exist  $k$  and  $k'$  with  $i \leq k \leq j$  and  $i' \leq k' \leq j'$  such that  $v^{[i,j]}$  returns the value  $v^{[k]}$  and  $v^{[i',j']}$  returns the value  $v^{[k']}$ .

This condition also asserts the same intuitive requirement that two consecutive reads obtain correctly-ordered values, but it does not imply atomicity. As a counterexample, let  $v^{[0]} = v^{[2]} = 0$  and  $v^{[1]} = 1$ , let  $R_1, R_2, R_3$  be the three reads shown in Figure 8, and suppose that  $R_1$  and  $R_3$  return the value 1 while  $R_2$  returns the value 0. (Since each of the reads overlaps a write that changes the value, they all see traces of both values and could return either of them.) The reader (of this paper) can show that this register is regular, but no such  $\phi$  can be constructed; there is no way to interpret these reads and writes as belonging to an atomic register while maintaining the given orderings among the writes and among the reads.

Let us now consider what happens if a global-time model exists. An atomic register is one in which reads and writes do not overlap. Both reads

and writes can then be shrunk to a point—that is, reduced to arbitrarily small time intervals within the interval in which they actually occur. For a regular register, it is shown in [5] that reads may be shrunk to a point, so each read overlaps at most one write. However, for a regular register that is not atomic, not all writes can be shrunk to a point.

If two reads cannot overlap the same write, then  $v^{[i,j]} \longrightarrow v^{[i',j']}$  implies  $j \leq i'$ . This implies that any  $\phi$  satisfying conditions 1 and 2 of Proposition 5 also satisfies condition 3. But such a  $\phi$  exists if  $v$  is regular, so any regular register trivially implements an atomic one if two reads cannot overlap a single write.

## 7 Correctness Proofs for the Constructions

### 7.1 Proof of Constructions 1, 2, and 3

These constructions are all simple, and the correctness proofs are essentially trivial. Formal proofs add no further insight into the constructions, but they do illustrate how the formalism of Part I and the register axioms of the preceding section are applied to actual algorithms. Therefore all the formal details in the proof of Construction 1 are indicated, while the formal proofs for the other two constructions are just briefly sketched.

Recall that in Construction 1, the  $m$ -reader register  $v$  is implemented by the  $m$  single-reader registers  $v_i$ . Formally, this construction defines a system, denoted by  $\mathbf{S}$ , that is the set of all system executions consisting of reads and writes of the  $v_i$  such that the only operations to these registers are the ones indicated by the readers' and writer's programs. Thus,  $\mathbf{S}$  contains all system executions  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  such that:

- $\mathcal{S}$  consists of reads and writes of the registers  $v_i$ .
- Each  $v_i$  is written by the same writer and is read only by the  $i^{\text{th}}$  reader.
- For any  $i$  and  $j$ : if the write  $V_i^{[k]}$  occurs, then the write  $V_j^{[k]}$  also occurs and  $V_i^{[k-1]} \longrightarrow V_j^{[k]}$ .

The third condition expresses the formal semantics of the writer's algorithm, asserting that a write of  $v$  is done by writing all the  $v_i$ , and that a write of  $v$  is completed before the next one is begun.

To say that the  $v_i$  are safe or regular means that the system  $\mathbf{S}$  is further restricted to contain only system executions that satisfy B0–B3 or B0–B4, when each  $v_i$  is substituted for  $v$  in those conditions.

According to Definition 8 of Part I, showing that this construction implements a register  $v$  requires constructing a mapping  $\iota$  from  $\mathbf{S}$  to the system  $\mathbf{H}$ , the latter system consisting of the set of all system executions formed by reads and writes to an  $m$ -reader register  $v$ . To say that  $v$  is safe or regular means that  $\mathbf{H}$  contains only system executions satisfying B0–B3 or B0–B4.

In giving the readers' and writer's algorithms, the construction implies that, for each system execution  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  of  $\mathbf{S}$ , the set  $\iota(\mathcal{S})$  of operation executions of  $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$  is the higher-level view of  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  consisting of all writes  $V^{[k]}$  of the form  $\{V_1^{[k]}, \dots, V_m^{[k]}\}$ , for  $V_i^{[k]} \in \mathcal{S}$ , and all reads of the form  $\{R_i\}$ , where  $R_i \in \mathcal{S}$  is a read of  $v_i$ . (The write  $V^{[k]}$  exists in  $\iota(\mathcal{S})$  if and only if some, and hence all,  $V_i^{[k]}$  exist.) Conditions H1 and H2 of Definition 4 in Part I are obviously satisfied, so this is indeed a higher-level view. To complete the mapping  $\iota$ , we must define the precedence relations  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$  so that  $\iota(\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle)$  is defined to be  $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ . Proving the correctness of the construction means showing that:

1.  $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$  is a system execution. This requires proving that A1–A5 are satisfied.
2.  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$  implements  $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ . This requires proving that H1–H3 are satisfied.
3.  $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$  is in  $\mathbf{H}$ . This requires proving that B0–B3 or B0–B4 are satisfied.

The precedence relations on  $\iota(\mathcal{S})$  are defined to be the “real” ones, with  $G \xrightarrow{\mathcal{H}} H$  if and only if  $G$  really precedes  $H$ . Formally, this means that we let  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$  be the induced relations  $\xrightarrow{*}$  and  $\dashrightarrow^{*}$ , defined by equations (2) in Section 2 of Part I. It was pointed out in that section that the induced precedence relations make any higher-level view a system execution, so 1 is satisfied. It was already observed that H1 and H2, which are independent of the choice of precedence relations, are satisfied, and H3 is trivially satisfied by the induced precedence relations, so 2 holds. Therefore, it suffices to show that, if B0–B3 or B0–B4 are satisfied for reads and writes of each of the registers  $v_i$  in  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ , then they are also satisfied by the register  $v$  of  $\langle \iota(\mathcal{S}), \xrightarrow{\mathcal{H}}, \dashrightarrow^{\mathcal{H}} \rangle$ .

Properties B0 and B1 for  $\langle \iota(\mathcal{S}), \xrightarrow{*}, \dashrightarrow^{*} \rangle$  follow easily from equations (2) of Part I and the corresponding property for  $\langle \mathcal{S}, \rightarrow, \dashrightarrow \rangle$ . Property B2 is immediate. The informal proof of B3 is as follows: if a read of  $v$  by process  $i$

does not overlap a write (in  $\iota(\mathcal{S})$ ), then the read of  $v_i$  does not overlap any write of  $v_i$ , so it obtains the correct value. A formal proof is based upon:

- X. If a read  $R_i$  in  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  sees  $v_i^{[k,l]}$ , then the corresponding read  $\{R_i\}$  in  $\langle \iota(\mathcal{S}), \xrightarrow{*}, \dashrightarrow^* \rangle$  sees  $v^{[k',l']}$ , where  $k' \leq k \leq l \leq l'$ .

The proof of property X is a straightforward application of (2) of Part I and Definition 9. Property X implies that if B3 or B4 holds for  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$ , then it holds for  $\langle \iota(\mathcal{S}), \xrightarrow{*}, \dashrightarrow^* \rangle$ . This completes the formal proof of Construction 1.

The formal proof of Construction 2 is quite similar. Again, the induced precedence relations are used to turn a higher-level view into a system execution. The proof of Construction 3 is a bit trickier because a write operation to  $v^*$  that does not change its value consists only of the read operation to the internal variable  $x$ . This means that the induced precedence relation  $\dashrightarrow^*$  does not necessarily satisfy B1, so  $\longrightarrow$  and  $\dashrightarrow^*$  must be extended to relations  $\xrightarrow{\mathcal{H}}$  and  $\dashrightarrow^{\mathcal{H}}$  for which B1 holds. This is done as follows. For every read-write pair  $R, W$  for which neither  $R \dashrightarrow^* W$  nor  $W \dashrightarrow^* R$  holds, add either one of the relations  $R \dashrightarrow^{\mathcal{H}} W$  or  $W \dashrightarrow^{\mathcal{H}} R$  (it does not matter which), and then add all the extra relations implied by A3, A4, and the transitivity of  $\xrightarrow{\mathcal{H}}$ . It is then necessary to show that the new precedence relations satisfy A1–A5, the only nontrivial part being the proof that  $\xrightarrow{\mathcal{H}}$  is acyclic. Alternatively, one can simply apply Proposition 3 of [5], which asserts the existence of the required precedence relations.

## 7.2 Proof of Construction 4

The higher-level system execution of reads and writes to  $v$  is defined to have the induced precedence relations  $\xrightarrow{*}$  and  $\dashrightarrow^*$ . As in the above proofs, verifying that this defines an implementation and that B0 and B1 hold is trivial. The only problems are proving B2—namely, showing that the reader must find some  $v_i$  equal to one—and proving B4 (which implies B3).

First, the following property is proved:

- Y. If a read sees  $v^{[l,r]}$  and returns the value  $\mu$ , then there is some  $k$  with  $l \leq k \leq r$  such that  $v^{[k]} = \mu$ .

If B2 holds, then property Y implies B4.

Reasoning about the construction is complicated by the fact that a write of  $v$  does not write all the  $v_j$ , so the write of  $v_j$  that occurs during the  $k^{\text{th}}$

write of  $v$  is not necessarily the  $k^{\text{th}}$  write of  $v_j$ . To overcome this difficulty, new names for the write operations to the  $v_j$  are introduced. If  $v_j$  is written during the execution of  $V^{[k]}$ , then  $W_j^{[k]}$  denotes that write of  $v_j$ ; otherwise,  $W_j^{[k]}$  is undefined. Thus, every write  $V_j^{[l]}$  of  $v_j$  is also named  $W_j^{[l']}$  for some  $l' \geq l$ . A read of  $v_j$  is said to see  $w_j^{[l',r']}$  if it sees  $v_j^{[l,r]}$  and the writes  $W_j^{[l']}$  and  $W_j^{[r']}$  are the same writes as  $V_j^{[l]}$  and  $V_j^{[r]}$ , respectively. Note that, because the writer's algorithm writes from "right to left",  $W_1^{[k]}$  exists for all  $k$  and, if  $W_i^{[k]}$  exists, then so do all the  $W_j^{[k]}$  with  $j < i$ .

Let  $R$  be a read that returns the value  $\mu$ , and let  $\mu$  be the  $i^{\text{th}}$  value, so  $R$  consists of the sequence of reads  $R_1 \longrightarrow \dots \longrightarrow R_i$ , where each  $R_j$  is a read of  $v_j$ . All the  $R_j$  return the value 0 except  $R_i$ , which returns the value 1. Let  $R$  see  $v^{[l,r]}$  and let each  $R_j$  see  $w_j^{[l(j),r(j)]}$ . By regularity of  $v_j$ , there is some  $k(j)$  with  $l(j) \leq k(j) \leq r(j)$  such that  $W_i^{[k(i)]}$  writes a 1 and  $W_j^{[k(j)]}$  writes a 0 for  $1 \leq j < i$ . Thus,  $v^{[k(i)]}$  is the value read by  $R$ , so it suffices to show that  $l \leq k(i) \leq r$ .

Definition 9 applied to the read  $R_i$  of  $v$  implies  $W_i^{[r(i)]} \dashrightarrow R_i$ , which, by equation (2) of Part I, implies  $V^{[r(i)]} \dashrightarrow^* R$ . This in turn implies  $r(i) \leq r$ , so  $k(i) \leq r$ .

For any  $p$  with  $p \leq l$ , Definition 9 implies that  $R \dashrightarrow V^{[p]}$ , which implies that  $R_1 \dashrightarrow W_1^{[p]}$ , which in turn implies that  $p \leq l(1)$ . Hence, letting  $p = l$ , we have  $l \leq l(1)$ .<sup>10</sup> Since  $l(j) \leq k(j)$ , it suffices to prove that  $k(j) \leq l(j+1)$  for  $1 \leq j < i$ .

Since  $k(j) \leq r(j)$ , Definition 9 implies that  $W_j^{[k(j)]} \dashrightarrow R_j$ . Because  $W_j^{[k(j)]}$  writes a zero,  $W_{j+1}^{[k(j)]}$  exists, and we have

$$W_{j+1}^{[k(j)]} \longrightarrow W_j^{[k(j)]} \dashrightarrow R_j \longrightarrow R_{j+1}$$

where the two  $\longrightarrow$  relations are implied by the order in which writing and reading of the individual  $v_j$  are performed. By A4, this implies that  $W_{j+1}^{[k(j)]} \longrightarrow R_{j+1}$ , which, by A2, implies  $R_{j+1} \dashrightarrow W_{j+1}^{[k(j)]}$ . By Definition 9, this implies that  $k(j) \leq l(j+1)$ , completing the proof of property Y.

To complete the proof of the construction, it suffices to prove that every read does return a value. Let  $R$  and the values  $l(j)$ ,  $k(j)$ , and  $r(j)$  be as

---

<sup>10</sup>Note that the same argument does not prove that  $l \leq l(i)$  because  $W_i^{[p]}$  does not necessarily exist.

above, except let  $i = n$  and drop the assumption that  $R_i$  obtains the value 1. To prove B2, it is necessary to prove that  $R_n$  does obtain the value 1.

The same argument used above shows that, if  $R_j$  obtains a zero, then that zero was written by some write  $W_j^{[k(j)]}$ , which implies that  $W_{j+1}^{[k(j)]}$  exists and  $k(j) \leq l(j+1)$ . Since  $R_n$  obtains the value written by  $W_n^{[k(n)]}$ , it must obtain a 1 unless  $k(n) = 0$  and the initial value is not the  $n^{\text{th}}$  one. Suppose the initial value  $v^{[0]}$  is the  $p^{\text{th}}$  value, encoded with  $v_p = 1$ ,  $p < n$ . Since  $R_p$  obtains the value 0, we must have  $k(p) > 0$ , which implies that  $k(n) > 0$ , so  $R_n$  obtains the value 1. This completes the proof of the construction.

### 7.3 Proof of Construction 5

This construction defines a set  $\mathcal{H}$ , consisting of reads and writes of  $v^*$ , that is a higher-level view of a system execution  $\langle \mathcal{S}, \longrightarrow, \dashrightarrow \rangle$  whose operation executions are reads and writes of the two shared registers  $v$  and  $c$ . As usual,  $\xrightarrow{*}$  and  $\dashrightarrow^*$  denote the induced precedence relations on  $\mathcal{S}$  that are defined by (2) of Part I.

In this construction, the write  $V^{*[k+1]}$  of  $v^*$ , for  $k \geq 0$ , is implemented by the sequence

$$RC_k \longrightarrow V^{[3k+1]} \longrightarrow V^{[3k+2]} \longrightarrow V^{[3k+3]} \quad (3)$$

where  $\text{num}(v^{[3k+i]}) = i$  and  $RC_k$  is a read of  $c$  that obtains the value  $\neg \text{col}(v^{[3k+i]})$ , the colors  $\text{col}(v^{[3k+1]})$  being the same for the three values of  $i$ . (Recall that  $V^{[p]}$  is the  $p^{\text{th}}$  write of  $v$  and  $v^{[p]}$  is the value it writes.) The initial write  $V^{*[0]}$  of  $v^*$  is just the initial write  $V^{[0]}$  of  $v$ .

Since there is only one reader, the reads of  $v^*$  are totally ordered by  $\xrightarrow{*}$ . The  $j^{\text{th}}$  read  $R_j^*$  of  $v^*$  consists of the sequence  $RV_j \longrightarrow C^{[j]}$ , where  $RV_j$  is the  $j^{\text{th}}$  read of  $v$  and  $C^{[j]}$  is the  $j^{\text{th}}$  write of  $c$ .

The proof of correctness is based upon Proposition 5. Letting  $\phi(j)$  denote  $\phi(R_j^*)$ , to apply that proposition, it suffices to choose the  $\phi(j)$  such that the following three properties hold:

1. If  $R_j^*$  sees  $v^{*[l,r]}$  then  $l \leq \phi(j) \leq r$ .
2.  $R_j^*$  returns the value  $v^{*\phi(j)}$ .
3. If  $j' < j$  then  $\phi(j') \leq \phi(j)$ .

Intuitively, the existence of such a function  $\phi$  means we can pretend that the read  $R_j^*$  occurred after the  $\phi(j)^{\text{th}}$  write and before the  $\phi(j) + 1^{\text{st}}$  write of  $v^*$ .

To construct such a  $\phi$ , a function  $\psi$  is first defined such that  $RV_j$  returns the value  $v^{[\psi(j)]}$  and, if  $RV_j$  sees  $v^{[l,r]}$ , then  $l \leq \psi(j) \leq r$ . Since  $v$  is regular, such a  $\psi$  exists. From part (c) of Proposition 3, we have:

$$j' < j \text{ implies } \psi(j') \leq \psi(j) + 1 \quad (4)$$

We define  $\phi(j)$  as follows. If  $\psi(j) = 3k + i$ , with  $1 \leq i \leq 3$ , then  $\phi(j)$  equals  $k$  if  $R_j^*$  returns the value  $old(rv)$  (by executing the innermost **else** clause of the reader's algorithm) and it equals  $k + 1$  if  $R_j^*$  returns the value  $new(rv)$ . We must now prove that  $\phi$  satisfies properties 1–3.

By Proposition 4, to prove property 1 it suffices to prove:

$$V^{[\phi(j)]} \xrightarrow{*} R_j^* \xrightarrow{*} V^{[\phi(j)+1]} \quad (5)$$

Proposition 4 implies that

$$V^{[\psi(j)]} \dashrightarrow RV_j \dashrightarrow V^{[\psi(j)+1]} \quad (6)$$

If  $\psi(j) = 3k + 3$ , then  $V^{[\psi(j)]}$  is part of  $V^{[k+1]}$  and  $V^{[\psi(j)+1]}$  is part of  $V^{[k+2]}$ , so (6) and the definition of  $\dashrightarrow$  imply

$$V^{[k+1]} \xrightarrow{*} R_j^* \xrightarrow{*} V^{[k+2]}$$

But  $\psi(j) = 3k + 3$  implies that  $R_j^*$  obtains  $num(rv) = 3$  and therefore returns  $new(rv)$ , so, by definition of  $\phi$ ,  $\phi(j) = k + 1$ , which proves (5).

If  $\psi(j) = 3k + i$  with  $1 \leq i \leq 3$ , then  $V^{[\psi(j)]}$  and  $V^{[\psi(j)+1]}$  are both part of  $V^{[k+1]}$ , so (6) and the definition of  $\dashrightarrow$  imply

$$V^{[k+1]} \xrightarrow{*} R_j^* \xrightarrow{*} V^{[k+1]}$$

Since  $V^{[k]} \xrightarrow{*} V^{[k+1]} \xrightarrow{*} V^{[k+2]}$ , (5) follows from Axiom A3 when  $\phi(j)$  equals either  $k$  or  $k + 1$ , which, by definition of  $\phi$ , are the only two possibilities. This finishes the proof of (5), which proves property 1.

Property 2 follows immediately from the definition of  $\phi$  and the observation that if  $1 \leq i \leq 3$ , then  $v^{*[k]} = old(v^{[3k+i]})$  and  $v^{*[k+1]} = new(v^{[3k+i]})$ .

To prove property 3, it suffices to show that, for every  $j$ ,  $\phi(j-1) \leq \phi(j)$ . By (4),  $\psi(j-1) \leq \psi(j) + 1$ . It therefore follows from the definition of  $\phi$  that there are only two situations in which  $\phi(j-1)$  could be greater than  $\phi(j)$ :

- (a)  $\psi(j) = 3k + i$ ,  $1 \leq i \leq 3$ , and  $R_j^*$  returns  $old(rv)$ , and  $\psi(j-1) = 3k + i'$ ,  $1 \leq i' \leq 3$ , and  $R_{j-1}^*$  returns  $new(rv)$ .
- (b)  $\psi(j) = 3k$ ,  $\psi(j-1) = 3k + 1$ , and  $R_{j-1}^*$  returns  $new(rv)$ .

We first show that case (a) is impossible. Since  $\psi(j-1) \leq \psi(j) + 1$ , we have  $i' \leq i + 1$ . However,  $i$  is the value of  $num(rv)$  obtained by  $R_j^*$ , while  $i'$  is the value of  $num(rv)$  obtained by  $R_{j-1}^*$  and hence the value of  $num(rv')$  during the execution of  $R_j^*$  (after it executes the first assignment statement). Therefore, when executing  $R_j^*$ , the reader finds *nuret* true (because  $R_{j-1}^*$  returned  $new(rv)$ ),  $col(rv) = col(rv')$  (because both  $R_j^*$  and  $R_{j-1}^*$  obtained values written by the same write  $V^{*[k+1]}$ ), and  $num(rv) \geq num(rv') - 1$  (because  $i' \leq i + 1$ ). Hence  $R_j^*$  must return  $new(rv)$ , so case (a) is impossible.

Finally, we show the impossibility of case (b). This is the most difficult part of the proof, and essentially involves proving the assertion made in Section 5 that, if a read obtains the value  $(\mu, \nu, 1, \kappa)$  and returns the value  $\nu$ , then it and a preceding read both overlap a write of the value  $(\mu, \nu, 2, \kappa)$ .

Examination of the reader's algorithm reveals that for case (b) to occur, there must exist reads  $R_{j_3}^*$  and  $R_{j_2}^*$  such that (i)  $j_3 < j_2 < j - 1$ , (ii) each  $R_{j_i}^*$  obtains a value of  $rv$  with  $num(rv) = i$  and  $col(rv)$  equal to the value of  $col(rv)$  obtained by  $R_{j-1}^*$ , and (iii) every read between  $R_{j_3}^*$  and  $R_{j-1}^*$  also obtains the same value of  $col(rv)$  as  $R_{j-1}^*$ . For notational convenience, let  $j_1 = j - 1$  and let  $\kappa$  denote the value of  $col(rv)$  obtained by the reads  $R_{j_i}^*$ . We then have:

$$RV_{j_3} \longrightarrow C^{[j_3]} \longrightarrow RV_{j_2} \longrightarrow C^{[j_2]} \longrightarrow RV_{j_1} \longrightarrow C^{[j_1]} \quad (7)$$

$$j_3 \leq j \leq j_1 \text{ implies } c^{[j]} = \kappa \quad (8)$$

Since  $R_{j_i}^*$  obtains  $num(rv) = i$ ,  $\psi(j_i)$  equals  $3k_i + i$  for some  $k_i$ . Since  $R_{j_i}^*$  obtains  $col(rv) = \kappa$ ,  $RC_{k_i}$  reads the value  $\neg\kappa$ . (Remember that  $RC_k$  is the read of  $c$  that is part of the write  $V^{*[k+1]}$ ).

Since  $\psi(j_i) = 3k_i + i$ , substituting  $j_i$  for  $j$  in (6) yields

$$V^{[3k_i+i]} \dashrightarrow RV_{j_i} \quad (9)$$

$$RV_{j_i} \dashrightarrow V^{[3k_i+i+1]} \quad (10)$$

We show now that  $k_1 = k_2$ , which shows that  $R_{j_2}^*$  and  $R_{j_1}^*$  overlap the same write of  $v^*$ . The proof is by contradiction. First, assume that  $k_2 > k_1$ . This implies that  $V^{[3k_1+2]} \longrightarrow V^{[3k_2+2]}$ , which, with (7) and (10), yields

$$RV_{j_2} \longrightarrow RV_{j_1} \dashrightarrow V^{[3k_1+2]} \longrightarrow V^{[3k_2+2]}$$

Applying Axiom A4, we get  $RV_{j_2} \longrightarrow V^{[3k_2+2]}$ , and, by Axiom A2, this contradicts (9), so we must have  $k_2 \leq k_1$ .

Next, assume that  $k_2 < k_1$ . This implies that  $V^{[k_2+3]} \longrightarrow RC_{k_1}$ . Combining this with (7) and (10) gives

$$C^{[j_3]} \longrightarrow RV_{j_2} \dashrightarrow V^{[k_2+3]} \longrightarrow RC_{k_1}$$

and Axiom A4 implies

$$C^{[j_3]} \longrightarrow RC_{k_1} \tag{11}$$

Let  $l$  and  $r$  be integers such that  $RC_{k_1}$  sees  $c^{[l,r]}$ . By part (b) of Proposition 3, (11) implies that  $j_3 \leq l$ . Since  $RC_{k_1}$  obtains the value  $\neg\kappa$ , (8) and the regularity of  $c$  (Axiom B4) imply that  $r > j_1$ . Part (a) of Proposition 4 (substituting  $j_1 + 1$  for  $k$  and  $r$  for  $j$ ) then implies  $C^{[j_1+1]} \dashrightarrow RC_{k_1}$ . Since  $C^{[j_1+1]}$  is part of a later read operation execution than is  $RV_{j_1}$ , we have  $RV_{j_1} \longrightarrow C^{[j_1+1]}$ . Combining these two relations with (3) gives

$$RV_{j_1} \longrightarrow C^{[j_1+1]} \dashrightarrow RC_{k_1} \longrightarrow V^{[3k_1+1]}$$

which by A4 implies  $RV_{j_1} \longrightarrow V^{[3k_1+1]}$ . Axiom A2 and (9) imply that this is impossible, so we have the contradiction that completes the proof that  $k_1 = k_2$ .

Returning to (b), recall that  $j_1 = j - 1$  and  $k_1 = k$ . We have  $\psi(j) = 3k$ ,  $\psi(j_2) = 3k_2 + 2 = 3k + 2$ , and  $j_2 < j - 1 < j$ , which contradicts (4). Hence, this shows that (b) is impossible, which completes the proof of property 3, completing the proof of correctness of the construction.

## 8 Conclusion

I have defined three classes of shared registers for asynchronous interprocess communication and have provided algorithms for implementing stronger classes in terms of weaker ones. For single-writer registers, the only unsolved problem is implementing a multireader atomic register. A solution probably exists, but it undoubtedly requires that a reader communicate with all other readers as well as with the writer. Also, more efficient implementations than Constructions 4 and 5 probably exist. For multivalued registers, Peterson's algorithm [14] combined with Construction 5 provides a more efficient implementation of a regular register than Construction 4, and a more efficient implementation of a single-reader atomic register than Construction 5. However, in this solution, Construction 4 is still needed to implement the regular register used in Construction 5.

The only closely related work that I know of is that of Misra [13]. Misra’s main result is a generalization of a restricted version of Proposition 5 of Section 6. It generalizes the proposition to multiple writers, but assumes a global-time model rather than using the more general formalism of Part I.

I have not addressed the question of multiwriter shared registers. It is not clear what assumptions one should make about the effect of overlapping writes. The one case that is straightforward is that of an atomic multiwriter register—the kind of register traditionally assumed in shared-variable concurrent programs. This raises the problem of implementing a multiwriter atomic register from single-writer ones. An unpublished algorithm of Bard Bloom implements a two-writer atomic register using single-writer atomic registers.

The definitions and proofs have all employed the general formalism developed in Part I. Instead of the more traditional approach of considering starting and stopping times of the operation executions, this formalism is based upon two abstract precedence relations satisfying Axioms A1–A5. These axioms embody the fundamental properties of temporal relations among operation executions that are needed to analyze concurrent algorithms.

## **Acknowledgements**

The algorithms described here were derived many years ago, although they have never before appeared in print. Carel Scholten provided much of the intellectual stimulation that led to their discovery. Jay Misra is largely responsible for persuading me to finally publish them, and Fred Schneider helped make the current version less unreadable than previous ones.



## References

- [1] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.
- [2] W. Brauer, editor. *Net Theory and Applications*. Springer-Verlag, Berlin, 1980.
- [3] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with “readers” and “writers”. *Communications of the ACM*, 14(10):190–199, October 1971.
- [4] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.
- [5] Leslie Lamport. *Interprocess Communication*. Technical Report, SRI International, March 1985.
- [6] Leslie Lamport. The mutual exclusion problem. To appear in *JACM*.
- [7] Leslie Lamport. A new approach to proving the correctness of multi-process programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.
- [8] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [9] Leslie Lamport. What it means for a concurrent program to satisfy a specification: why no one has specified priority. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, New Orleans, January 1985.
- [10] Peter E. Lauer, Michael W. Shields, and Eike Best. *Formal Theory of the Basic COSY Notation*. Technical Report TR143, Computing Laboratory, University of Newcastle upon Tyne, 1979.
- [11] A. Mazurkiewicz. *Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach*. Technical Report 84–19, Institute of Applied Mathematics and Computer Science, University of Leiden, 1984.
- [12] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, 1980.

- [13] J. Misra. Axioms for memory access in asynchronous hardware systems. 1984. To appear in *ACM Transactions on Programming Languages and Systems*.
- [14] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.
- [15] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, November 1977.
- [16] Glynn Winskel. *Events in Computation*. PhD thesis, Edinburgh University, 1980.



## 6. Conclusion

It has been shown that the existence of a  $\sigma$ -coloration of a particular graph is a necessary and sufficient condition for the existence of a solution to the class-teacher timetable problem with unavailability constraints and preassigned meetings. The knowledge of this necessary and sufficient condition does not provide an efficient algorithm which can be applied to an arbitrary timetable problem in order to determine the existence of a solution. The necessary and sufficient condition does, however, show that existing graph coloring algorithms [1, 6, 14, 15, 16] may be applied to timetable problems with unavailability constraints and preassigned meetings.

*Acknowledgment.* The authors would like to thank the referees for their constructive criticism and helpful suggestions for the improvement of this paper.

Received July 1973; revised February 1974

### References

1. Corneil, D.G., and Graham, B. An algorithm for determining the chromatic number of a graph. *SIAM J. on Computing* 2, 4 (Dec. 1973), 311-318.
2. Csima, J., and Gotlieb, C.C. A computer method for constructing school time-tables. Presented at ACM 18th Ann. Conf., 1963.
3. Dempster, M.A.H. On the Gotlieb-Csima time-tabling algorithm. *Canadian J. Math.* 20, 103-119.
4. Dempster, M.A.H. Two algorithms for the time-table problem. In *Combinatorial Mathematics and Its Applications* (D.J.A. Welsh, Ed.), Academic Press, London, 1969, pp. 63-85.
5. De Werra, D. Construction of school timetables by flow methods. *Infor.* 1, 1, 12-22.
6. Formby, J.A. Computer procedure for bounding the chromatic number of a graph. In *Combinatorial Mathematics and Its Applications* (D.J.A. Welsh, Ed.), Academic Press, London, 1969, pp. 111-114.
7. Gotlieb, C.C. The construction of class-teacher time-tables. Proc. IFIP Congress 62, Munich, North Holland Pub. Co., Amsterdam, 1963, pp. 73-77.
8. Lions, J. Matrix reduction using the Hungarian method for the construction of school timetables. *Comm. ACM* 9, 5 (May 1966), 349-354.
9. Lions, J. A counter-example for Gotlieb's method for the construction of school timetables. *Comm. ACM* 9, 9 (Sept. 1966), Letters to the Editor, 697-698.
10. Lions, J. A generalization of a method for the construction of class/teacher timetables. *Inform. Proc.* 68, Proc. IFIP Congress 1968, North Holland Pub. Co., Amsterdam, pp. 1377-1382.
11. Lions, J. The Ontario school scheduling program. *Computer J.* 10, (1967-68), 14-21.
12. Lions, J. Some results concerning the reduction of binary matrices. *J. ACM* 18, 3 (July 1971), 424-430.
13. Neufeld, G.A., and Tartar, J. Generalized graph colorations. *SIAM J. of Applied Math* (To appear).
14. Peck, J.E.L., and Williams, M.R. Algorithm 286, exam scheduling. *Comm. ACM* 9, 6 (June 1966), 433-434.
15. Welsh, D.J.A., and Powell, M.B. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Computer J.* 10, (1967-68) 85-86.
16. Williams, M.R. The coloring of very large graphs. Combinatorial Structures and Their Applications, Proc. Calgary Internat. Conf. on Combinatorial Structures and Their Application. Gordon and Breach, Calgary, Canada, June 1969, pp. 477-478.

Computer  
Systems

G. Bell, D. Siewiorek,  
and S.H. Fuller, Editors

# A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport  
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate despite the failure of any individual component.

**Key Words and Phrases:** critical section, concurrent programming, multiprocessing, semaphores

**CR Categories:** 4.32

## Introduction

Knuth [1], deBruijn [2], and Eisenberg and McGuire [3] have given solutions to a concurrent programming problem originally proposed and solved by Dijkstra [4]. A simpler solution using semaphores has also been implemented [5]. These solutions have one drawback for use in a true multicomputer system (rather than a time-shared multiprocessor system): the failure of a single unit will halt the entire system. We present a simple solution which allows the system to continue to operate despite the failure of any individual component.

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by U.S. Army Research Office—Durham, under Contract No. DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, MA 01880.

## The Algorithm

Consider  $N$  asynchronous computers communicating with each other only via shared memory. Each computer runs a cyclic program with two parts—a *critical section* and a *noncritical section*. Dijkstra's problem, as extended by Knuth, is to write the programs so that the following conditions are satisfied:

1. At any time, at most one computer may be in its critical section.
2. Each computer must eventually be able to enter its critical section (unless it halts).
3. Any computer may halt in its noncritical section.

Moreover, no assumptions can be made about the running speeds of the computers.

The solutions of [1–4] had all  $N$  processors set and test the value of a single variable  $k$ . Failure of the memory unit containing  $k$  would halt the system. The use of semaphores also implies reliance upon a single hardware component.

Our solution assumes  $N$  processors, each containing its own memory unit. A processor may read from any other processor's memory, but it need only write into its own memory. The algorithm has the remarkable property that if a read and a write operation to a single memory location occur simultaneously, then only the write operation must be performed correctly. The read may return *any* arbitrary value!

A processor may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts. There may then be a period when reading from its memory gives arbitrary values. Eventually, any read from its memory must give a value of zero. (In practice, a failed computer might be detected by its failure to respond to a read request within a specified length of time.)

Unlike the solutions of [1–4], ours is a first-come-first-served method in the following sense. When a processor wants to enter its critical section, it first executes a loop-free block of code—i.e. one with a fixed number of execution steps. It is then guaranteed to enter its critical section before any other processor which later requests service.

The algorithm is quite simple. It is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store. The holder of the lowest number is the next one served. In our algorithm, each processor chooses its own number. The processors are named  $1, \dots, N$ . If two processors choose the same number, then the one with the lowest name goes first.

The common store consists of

integer array *choosing* [1:N], *number* [1:N]

Words *choosing* ( $i$ ) and *number* ( $i$ ) are in the memory of processor  $i$ , and are initially zero. The range of values of *number* ( $i$ ) is unbounded. This will be discussed below.

The following is the program for processor  $i$ . Execution must begin inside the noncritical section. The argu-

ments of the maximum function can be read in any order. The relation "less than" on ordered pairs of integers is defined by  $(a,b) < (c,d)$  if  $a < c$ , or if  $a = c$  and  $b < d$ .

```

begin integer j ;
L1: choosing [i] := 1;
 number [i] := 1 + maximum (number [1], ..., number [N]);
 choosing [i] := 0;
 for $j = 1$ step 1 until N do
 begin
 L2: if choosing [j] ≠ 0 then goto L2;
 L3: if number [j] ≠ 0 and (number [j], j) < (number [i],
 i) then goto L3;
 end;
 critical section;
 number [i] := 0;
 noncritical section;
 goto L1;
end

```

We allow processor  $i$  to fail at any time, and then to be restarted in its noncritical section (with *choosing* [ $i$ ] = *number* [ $i$ ] = 0). However, if a processor keeps failing and restarting, then it can deadlock the system.

## Proof of Correctness

To prove the correctness of the algorithm, we first make the following definitions. Processor  $i$  is said to be *in the doorway* while *choosing* [ $i$ ] = 1. It is said to be *in the bakery* from the time it resets *choosing* ( $i$ ) to zero until it either fails or leaves its critical section. The correctness of the algorithm is deduced from the following assertions. Note that the proofs make no assumptions about the value read during an overlapping read and write to the same memory location.

*Assertion 1.* If processors  $i$  and  $k$  are in the bakery and  $i$  entered the bakery before  $k$  entered the doorway, then *number* [ $i$ ] < *number* [ $k$ ].

*Proof.* By hypothesis, *number* [ $i$ ] had its current value while  $k$  was choosing the current value of *number* [ $k$ ]. Hence,  $k$  must have chosen *number* [ $k$ ]  $\geq 1 + \text{number}$  [ $i$ ].  $\square$

*Assertion 2.* If processor  $i$  is in its critical section, processor  $k$  is in the bakery, and  $k \neq i$ , then  $(\text{number}$  [ $i$ ],  $i) < (\text{number}$  [ $k$ ],  $k)$ .

*Proof.* Since *choosing* [ $k$ ] has essentially just two values—zero and nonzero—we can assume that from processor  $i$ 's point of view, reading or writing it is done instantaneously, and a simultaneous read and write does not occur. For example, if *choosing* [ $k$ ] is being changed from zero to one while it is also being read by processor  $i$ , then the read is considered to happen first if it obtains a value of zero; otherwise the write is said to happen first. All times defined in the proof are from processor  $i$ 's viewpoint.

Let  $t_{L2}$  be the time at which processor  $i$  read *choosing* [ $k$ ] during its last execution of  $L2$  for  $j = k$ , and let  $t_{L3}$  be the time at which  $i$  began its last execution of  $L3$  for  $j = k$ , so  $t_{L2} < t_{L3}$ . When processor  $k$  was choosing its

current value of *number* [*k*], let *t<sub>e</sub>* be the time at which it entered the doorway, *t<sub>w</sub>* the time at which it finished writing the value of *number* [*k*], and *t<sub>c</sub>* the time at which it left the doorway. Then  $t_e < t_w < t_c$ .

Since *choosing* [*k*] was equal to zero at time *t<sub>L2</sub>*, we have either (a)  $t_{L2} < t_e$  or (b)  $t_c < t_{L2}$ . In case (a), Assertion 1 implies that *number* [*i*] < *number* [*k*], so the assertion holds.

In case (b), we have  $t_w < t_c < t_{L2} < t_{L3}$ , so  $t_w < t_{L3}$ . Hence, during the execution of statement *L3* begun at time *t<sub>L3</sub>*, processor *i* read the current value of *number* [*k*]. Since *i* did not execute *L3* again for *j* = *k*, it must have found  $(\text{number } [i], i) < (\text{number } [k], k)$ . Hence, the assertion holds in this case, too.  $\square$

*Assertion 3.* Assume that only a bounded number of processor failures may occur. If no processor is in its critical section and there is a processor in the bakery which does not fail, then some processor must eventually enter its critical section.

*Proof.* Assume that no processor ever enters its critical section. Then there will be some time after which no more processors enter or leave the bakery. At this time, assume that processor *i* has the minimum value of  $(\text{number } [i], i)$  among all processors in the bakery. Then processor *i* must eventually complete the **for** loop and enter its critical section. This is the required contradiction.  $\square$

*Assertion 2* implies that at most one processor can be in its critical section at any time. Assertions 1 and 2 prove that processors enter their critical sections on a first-come-first-served basis. Hence, an individual processor cannot be blocked unless the entire system is deadlocked. Assertion 3 implies that the system can only be deadlocked by a processor halting in its critical section, or by an unbounded sequence of processor failures and re-entries. The latter can tie up the system as follows. If processor *j* continually fails and restarts, then with bad luck processor *i* could always find *choosing* [*j*] = 1, and loop forever at *L2*.

### Further Remarks

If there is always at least one processor in the bakery, then the value of *number* [*i*] can become arbitrarily large. This problem cannot be solved by any simple scheme of cycling through a finite set of integers. For example, given any numbers *r* and *s*, if  $N \geq 4$ , then it is possible to have simultaneously *number* (*i*) = *r* and *number* (*j*) = *s* for some *i* and *j*.

Fortunately, practical considerations will place an upper bound on the value of *number* [*i*] in any real application. For example, if processors enter the doorway at the rate of at most one per msec, then after a year of operation we will have  $\text{number } [i] < 2^{35}$ —assuming that a read of *number* [*i*] can never obtain a value larger than one which has been written there.

The unboundedness of *number* [*i*] does raise an inter-

<sup>1</sup> We have recently found such an algorithm, but it is quite complicated.

esting theoretical question: can one find an algorithm for finite processors such that processors enter their critical sections on a first-come-first-served basis, and no processor may write into another processor's memory? The answer is not known.<sup>1</sup>

The algorithm can be generalized in two ways: (i) under certain circumstances, to allow two processors simultaneously to be in their critical sections; and (ii) to modify the first-come-first-served property so that higher priority processors are served first. This will be described in a future paper.

### Conclusion

Our algorithm provides a new, simple solution to the mutual exclusion problem. Since it does not depend upon any form of central control, it is less sensitive to component failure than previous solutions.

Received September 1973; revised January 1974

### References

1. Knuth, D.E. Additional comments on a problem in concurrent programming control. *Comm. ACM* 9, 5 (May 1966), 321–322.
2. deBruijn, N.G. Additional comments on a problem in concurrent programming control *Comm. ACM* 10, 3 (Mar. 1967), 137–138.
3. Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. *Comm. ACM* 15, 11 (Nov. 1972), 999.
4. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (Sept. 1965), 569.
5. Dijkstra, E.W. The structure of THE multiprogramming system. *Comm. ACM* 11, 5 (May 1968), 341–346.

---

### Computer Systems

### Erratum

In "A Note on Subexpression Ordering in the Evaluation of Arithmetic Expressions" by Peter J. Denning and G. Scott Graham, *Comm. ACM* 16, 11 (Nov. 1973), 700–702, the following erratum has been submitted by Denning.

The first two sentences in the first full paragraph on p. 701 should read as follows:

Hu shows that an optimal list *L<sub>0</sub>* for any *m* and any tree (of equal-execution-time tasks) can be constructed by taking a first appearance of each task in the sequence  $M_{11}, M_{22}, \dots, M_{KK}$ . Ramamoorthy and Gonzales order the tasks of each  $M_{ij}$  according to decreasing execution time, then construct a list *L* by taking the first appearance of each task in the sequence  $M_{11}, \dots, M_{1K}, M_{22}, \dots, M_{2K}, M_{33}, \dots, M_{3K}, \dots, M_{KK}$ ; they claim that *L* is optimal for any tree and any *m*.

It should be noted that even for equal-execution-time tasks, a list constructed from the latter sequence above need not be consistent with the former sequence above and, hence, need not be optimal for that reason alone.

We are grateful to Dr. Shimon Even for calling our unfortunately incorrect wording to our attention.



# Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport  
Massachusetts Computer Associates, Inc.

**The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.**

**Key Words and Phrases:** distributed systems, computer networks, clock synchronization, multiprocess systems

**CR Categories:** 4.32, 5.29

## Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.

© 1978 ACM 0001-0782/78/0700-0558 \$00.75

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocesssing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

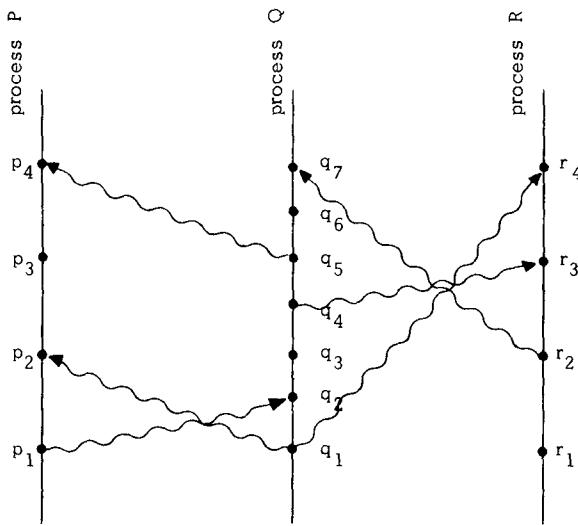
In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

## The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

Fig. 1.



event. We are assuming that the events of a process form a sequence, where  $a$  occurs before  $b$  in this sequence if  $a$  happens before  $b$ . In other words, a single process is defined to be a set of events with an a priori total ordering. This seems to be what is generally meant by a process.<sup>1</sup> It would be trivial to extend our definition to allow a process to split into distinct subprocesses, but we will not bother to do so.

We assume that sending or receiving a message is an event in a process. We can then define the “happened before” relation, denoted by “ $\rightarrow$ ”, as follows.

**Definition.** The relation “ $\rightarrow$ ” on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If  $a$  and  $b$  are events in the same process, and  $a$  comes before  $b$ , then  $a \rightarrow b$ . (2) If  $a$  is the sending of a message by one process and  $b$  is the receipt of the same message by another process, then  $a \rightarrow b$ . (3) If  $a \rightarrow b$  and  $b \rightarrow c$  then  $a \rightarrow c$ . Two distinct events  $a$  and  $b$  are said to be *concurrent* if  $a \not\rightarrow b$  and  $b \not\rightarrow a$ .

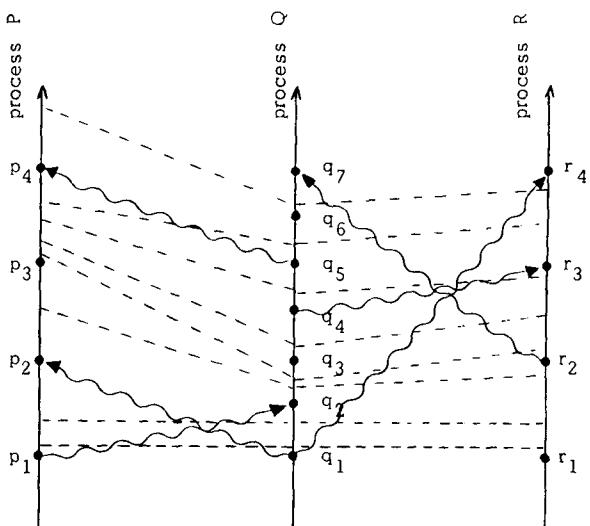
We assume that  $a \not\rightarrow a$  for any event  $a$ . (Systems in which an event can happen before itself do not seem to be physically meaningful.) This implies that  $\rightarrow$  is an irreflexive partial ordering on the set of all events in the system.

It is helpful to view this definition in terms of a “space-time diagram” such as Figure 1. The horizontal direction represents space, and the vertical direction represents time—later times being higher than earlier ones. The dots denote events, the vertical lines denote processes, and the wavy lines denote messages.<sup>2</sup> It is easy to see that  $a \rightarrow b$  means that one can go from  $a$  to  $b$  in

<sup>1</sup> The choice of what constitutes an event affects the ordering of events in a process. For example, the receipt of a message might denote the setting of an interrupt bit in a computer, or the execution of a subprogram to handle that interrupt. Since interrupts need not be handled in the order that they occur, this choice will affect the ordering of a process’ message-receiving events.

<sup>2</sup> Observe that messages may be received out of order. We allow the sending of several messages to be a single event, but for convenience we will assume that the receipt of a single message does not coincide with the sending or receipt of any other message.

Fig. 2.



the diagram by moving forward in time along process and message lines. For example, we have  $p_1 \rightarrow r_4$  in Figure 1.

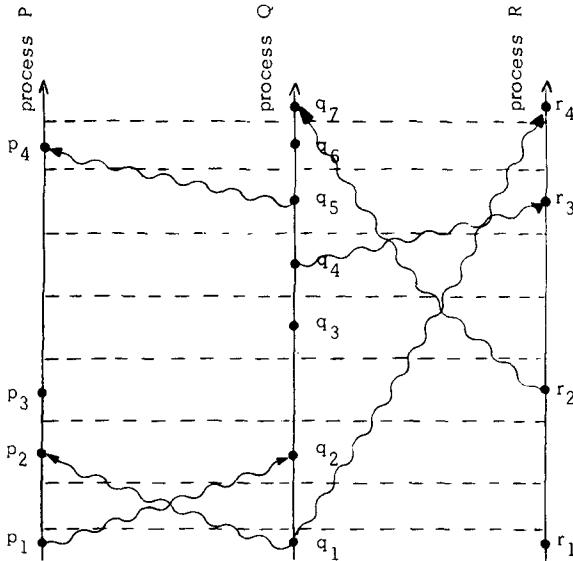
Another way of viewing the definition is to say that  $a \rightarrow b$  means that it is possible for event  $a$  to causally affect event  $b$ . Two events are concurrent if neither can causally affect the other. For example, events  $p_3$  and  $q_3$  of Figure 1 are concurrent. Even though we have drawn the diagram to imply that  $q_3$  occurs at an earlier physical time than  $p_3$ , process P cannot know what process Q did at  $q_3$  until it receives the message at  $p_4$ . (Before event  $p_4$ , P could at most know what Q was *planning* to do at  $q_3$ .)

This definition will appear quite natural to the reader familiar with the invariant space-time formulation of special relativity, as described for example in [1] or the first chapter of [2]. In relativity, the ordering of events is defined in terms of messages that *could* be sent. However, we have taken the more pragmatic approach of only considering messages that actually *are* sent. We should be able to determine if a system performed correctly by knowing only those events which *did* occur, without knowing which events *could* have occurred.

### Logical Clocks

We now introduce clocks into the system. We begin with an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, we define a clock  $C_i$  for each process  $P_i$  to be a function which assigns a number  $C_i(a)$  to any event  $a$  in that process. The entire system of clocks is represented by the function  $C$  which assigns to any event  $b$  the number  $C(b)$ , where  $C(b) = C_i(b)$  if  $b$  is an event in process  $P_i$ . For now, we make no assumption about the relation of the numbers  $C_i(a)$  to physical time, so we can think of the clocks  $C_i$  as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

Fig. 3.



We now consider what it means for such a system of clocks to be correct. We cannot base our definition of correctness on physical time, since that would require introducing clocks which keep physical time. Our definition must be based on the order in which events occur. The strongest reasonable condition is that if an event  $a$  occurs before another event  $b$ , then  $a$  should happen at an earlier time than  $b$ . We state this condition more formally as follows.

*Clock Condition.* For any events  $a, b$ :

$$\text{if } a \rightarrow b \text{ then } C(a) < C(b).$$

Note that we cannot expect the converse condition to hold as well, since that would imply that any two concurrent events must occur at the same time. In Figure 1,  $p_2$  and  $p_3$  are both concurrent with  $q_3$ , so this would mean that they both must occur at the same time as  $q_3$ , which would contradict the Clock Condition because  $p_2 \rightarrow p_3$ .

It is easy to see from our definition of the relation “ $\rightarrow$ ” that the Clock Condition is satisfied if the following two conditions hold.

C1. If  $a$  and  $b$  are events in process  $P_i$ , and  $a$  comes before  $b$ , then  $C_i(a) < C_i(b)$ .

C2. If  $a$  is the sending of a message by process  $P_i$ , and  $b$  is the receipt of that message by process  $P_j$ , then  $C_i(a) < C_j(b)$ .

Let us consider the clocks in terms of a space-time diagram. We imagine that a process’ clock “ticks” through every number, with the ticks occurring between the process’ events. For example, if  $a$  and  $b$  are consecutive events in process  $P_i$  with  $C_i(a) = 4$  and  $C_i(b) = 7$ , then clock ticks 5, 6, and 7 occur between the two events. We draw a dashed “tick line” through all the like-numbered ticks of the different processes. The space-time diagram of Figure 1 might then yield the picture in Figure 2. Condition C1 means that there must be a tick line between any two events on a process line, and

condition C2 means that every message line must cross a tick line. From the pictorial meaning of  $\rightarrow$ , it is easy to see why these two conditions imply the Clock Condition.

We can consider the tick lines to be the time coordinate lines of some Cartesian coordinate system on space-time. We can redraw Figure 2 to straighten these coordinate lines, thus obtaining Figure 3. Figure 3 is a valid alternate way of representing the same system of events as Figure 2. Without introducing the concept of physical time into the system (which requires introducing physical clocks), there is no way to decide which of these pictures is a better representation.

The reader may find it helpful to visualize a two-dimensional spatial network of processes, which yields a three-dimensional space-time diagram. Processes and messages are still represented by lines, but tick lines become two-dimensional surfaces.

Let us now assume that the processes are algorithms, and the events represent certain actions during their execution. We will show how to introduce clocks into the processes which satisfy the Clock Condition. Process  $P_i$ ’s clock is represented by a register  $C_i$ , so that  $C_i(a)$  is the value contained by  $C_i$  during the event  $a$ . The value of  $C_i$  will change between events, so changing  $C_i$  does not itself constitute an event.

To guarantee that the system of clocks satisfies the Clock Condition, we will insure that it satisfies conditions C1 and C2. Condition C1 is simple; the processes need only obey the following implementation rule:

IR1. Each process  $P_i$  increments  $C_i$  between any two successive events.

To meet condition C2, we require that each message  $m$  contain a *timestamp*  $T_m$  which equals the time at which the message was sent. Upon receiving a message timestamped  $T_m$ , a process must advance its clock to be later than  $T_m$ . More precisely, we have the following rule.

IR2. (a) If event  $a$  is the sending of a message  $m$  by process  $P_i$ , then the message  $m$  contains a timestamp  $T_m = C_i(a)$ . (b) Upon receiving a message  $m$ , process  $P_j$  sets  $C_j$  greater than or equal to its present value and greater than  $T_m$ .

In IR2(b) we consider the event which represents the receipt of the message  $m$  to occur after the setting of  $C_j$ . (This is just a notational nuisance, and is irrelevant in any actual implementation.) Obviously, IR2 insures that C2 is satisfied. Hence, the simple implementation rules IR1 and IR2 imply that the Clock Condition is satisfied, so they guarantee a correct system of logical clocks.

### Ordering the Events Totally

We can use a system of clocks satisfying the Clock Condition to place a total ordering on the set of all system events. We simply order the events by the times

at which they occur. To break ties, we use any arbitrary total ordering  $\prec$  of the processes. More precisely, we define a relation  $\Rightarrow$  as follows: if  $a$  is an event in process  $P_i$  and  $b$  is an event in process  $P_j$ , then  $a \Rightarrow b$  if and only if either (i)  $C_i(a) < C_j(b)$  or (ii)  $C_i(a) = C_j(b)$  and  $P_i \prec P_j$ . It is easy to see that this defines a total ordering, and that the Clock Condition implies that if  $a \rightarrow b$  then  $a \Rightarrow b$ . In other words, the relation  $\Rightarrow$  is a way of completing the “happened before” partial ordering to a total ordering.<sup>3</sup>

The ordering  $\Rightarrow$  depends upon the system of clocks  $C_i$ , and is not unique. Different choices of clocks which satisfy the Clock Condition yield different relations  $\Rightarrow$ . Given any total ordering relation  $\Rightarrow$  which extends  $\rightarrow$ , there is a system of clocks satisfying the Clock Condition which yields that relation. It is only the partial ordering  $\rightarrow$  which is uniquely determined by the system of events.

Being able to totally order the events can be very useful in implementing a distributed system. In fact, the reason for implementing a correct system of logical clocks is to obtain such a total ordering. We will illustrate the use of this total ordering of events by solving the following version of the mutual exclusion problem. Consider a system composed of a fixed collection of processes which share a single resource. Only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict. We wish to find an algorithm for granting the resource to a process which satisfies the following three conditions: (I) A process which has been granted the resource must release it before it can be granted to another process. (II) Different requests for the resource must be granted in the order in which they are made. (III) If every process which is granted the resource eventually releases it, then every request is eventually granted.

We assume that the resource is initially granted to exactly one process.

These are perfectly natural requirements. They precisely specify what it means for a solution to be correct.<sup>4</sup> Observe how the conditions involve the ordering of events. Condition II says nothing about which of two concurrently issued requests should be granted first.

It is important to realize that this is a nontrivial problem. Using a central scheduling process which grants requests in the order they are received will not work, unless additional assumptions are made. To see this, let  $P_0$  be the scheduling process. Suppose  $P_1$  sends a request to  $P_0$  and then sends a message to  $P_2$ . Upon receiving the latter message,  $P_2$  sends a request to  $P_0$ . It is possible for  $P_2$ 's request to reach  $P_0$  before  $P_1$ 's request does. Condition II is then violated if  $P_2$ 's request is granted first.

To solve the problem, we implement a system of

<sup>3</sup> The ordering  $\prec$  establishes a priority among the processes. If a “fairer” method is desired, then  $\prec$  can be made a function of the clock value. For example, if  $C_i(a) = C_j(b)$  and  $j < i$ , then we can let  $a \Rightarrow b$  if  $j < C_i(a) \bmod N \leq i$ , and  $b \Rightarrow a$  otherwise; where  $N$  is the total number of processes.

<sup>4</sup> The term “eventually” should be made precise, but that would require too long a diversion from our main topic.

clocks with rules IR1 and IR2, and use them to define a total ordering  $\Rightarrow$  of all events. This provides a total ordering of all request and release operations. With this ordering, finding a solution becomes a straightforward exercise. It just involves making sure that each process learns about all other processes’ operations.

To simplify the problem, we make some assumptions. They are not essential, but they are introduced to avoid distracting implementation details. We assume first of all that for any two processes  $P_i$  and  $P_j$ , the messages sent from  $P_i$  to  $P_j$  are received in the same order as they are sent. Moreover, we assume that every message is eventually received. (These assumptions can be avoided by introducing message numbers and message acknowledgment protocols.) We also assume that a process can send messages directly to every other process.

Each process maintains its own *request queue* which is never seen by any other process. We assume that the request queues initially contain the single message  $T_0:P_0$  *requests resource*, where  $P_0$  is the process initially granted the resource and  $T_0$  is less than the initial value of any clock.

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process  $P_i$  sends the message  $T_m:P_i$  *requests resource* to every other process, and puts that message on its request queue, where  $T_m$  is the timestamp of the message.

2. When process  $P_j$  receives the message  $T_m:P_i$  *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to  $P_i$ .<sup>5</sup>

3. To release the resource, process  $P_i$  removes any  $T_m:P_i$  *requests resource* message from its request queue and sends a (timestamped)  $P_i$  *releases resource* message to every other process.

4. When process  $P_j$  receives a  $P_i$  *releases resource* message, it removes any  $T_m:P_i$  *requests resource* message from its request queue.

5. Process  $P_i$  is granted the resource when the following two conditions are satisfied: (i) There is a  $T_m:P_i$  *requests resource* message in its request queue which is ordered before any other request in its queue by the relation  $\Rightarrow$ . (To define the relation “ $\Rightarrow$ ” for messages, we identify a message with the event of sending it.) (ii)  $P_i$  has received a message from every other process timestamped later than  $T_m$ .<sup>6</sup>

Note that conditions (i) and (ii) of rule 5 are tested locally by  $P_i$ .

It is easy to verify that the algorithm defined by these rules satisfies conditions I-III. First of all, observe that condition (ii) of rule 5, together with the assumption that messages are received in order, guarantees that  $P_i$  has learned about all requests which preceded its current

<sup>5</sup> This acknowledgment message need not be sent if  $P_j$  has already sent a message to  $P_i$  timestamped later than  $T_m$ .

<sup>6</sup> If  $P_i \prec P_j$ , then  $P_i$  need only have received a message timestamped  $\geq T_m$  from  $P_j$ .

request. Since rules 3 and 4 are the only ones which delete messages from the request queue, it is then easy to see that condition I holds. Condition II follows from the fact that the total ordering  $\Rightarrow$  extends the partial ordering  $\rightarrow$ . Rule 2 guarantees that after  $P_i$  requests the resource, condition (ii) of rule 5 will eventually hold. Rules 3 and 4 imply that if each process which is granted the resource eventually releases it, then condition (i) of rule 5 will eventually hold, thus proving condition III.

This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*, consisting of a set  $C$  of possible commands, a set  $S$  of possible states, and a function  $e: C \times S \rightarrow S$ . The relation  $e(C, S) = S'$  means that executing the command  $C$  with the machine in state  $S$  causes the machine state to change to  $S'$ . In our example, the set  $C$  consists of all the commands  $P_i$  *requests resource* and  $P_i$  *releases resource*, and the state consists of a queue of waiting *request* commands, where the request at the head of the queue is the currently granted one. Executing a *request* command adds the request to the tail of the queue, and executing a *release* command removes a command from the queue.<sup>7</sup>

Each process independently simulates the execution of the State Machine, using the commands issued by all the processes. Synchronization is achieved because all processes order the commands according to their timestamps (using the relation  $\Rightarrow$ ), so each process uses the same sequence of commands. A process can execute a command timestamped  $T$  when it has learned of all commands issued by all other processes with timestamps less than or equal to  $T$ . The precise algorithm is straightforward, and we will not bother to describe it.

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, the resulting algorithm requires the active participation of all the processes. A process must know all the commands issued by other processes, so that the failure of a single process will make it impossible for any other process to execute State Machine commands, thereby halting the system.

The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail. We will just observe that the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events. A user can tell that a system has "crashed" only because he has been waiting too long for a response. A method which works despite the failure of individual processes or communication lines is described in [3].

<sup>7</sup> If each process does not strictly alternate *request* and *release* commands, then executing a *release* command could delete zero, one, or more than one request from the queue.

## Anomalous Behavior

Our resource scheduling algorithm ordered the requests according to the total ordering  $\Rightarrow$ . This permits the following type of "anomalous behavior." Consider a nationwide system of interconnected computers. Suppose a person issues a request  $A$  on a computer  $A$ , and then telephones a friend in another city to have him issue a request  $B$  on a different computer  $B$ . It is quite possible for request  $B$  to receive a lower timestamp and be ordered before request  $A$ . This can happen because the system has no way of knowing that  $A$  actually preceded  $B$ , since that precedence information is based on messages external to the system.

Let us examine the source of the problem more closely. Let  $\mathcal{S}$  be the set of all system events. Let us introduce a set of events which contains the events in  $\mathcal{S}$  together with all other relevant external events, such as the phone calls in our example. Let  $\rightarrow$  denote the "happened before" relation for  $\mathcal{S}$ . In our example, we had  $A \rightarrow B$ , but  $A \not\rightarrow B$ . It is obvious that no algorithm based entirely upon events in  $\mathcal{S}$ , and which does not relate those events in any way with the other events in  $\mathcal{S}$ , can guarantee that request  $A$  is ordered before request  $B$ .

There are two possible ways to avoid such anomalous behavior. The first way is to explicitly introduce into the system the necessary information about the ordering  $\rightarrow$ . In our example, the person issuing request  $A$  could receive the timestamp  $T_A$  of that request from the system. When issuing request  $B$ , his friend could specify that  $B$  be given a timestamp later than  $T_A$ . This gives the user the responsibility for avoiding anomalous behavior.

The second approach is to construct a system of clocks which satisfies the following condition.

*Strong Clock Condition.* For any events  $a, b$  in  $\mathcal{S}$ :  
if  $a \rightarrow b$  then  $C(a) < C(b)$ .

This is stronger than the ordinary Clock Condition because  $\rightarrow$  is a stronger relation than  $\Rightarrow$ . It is not in general satisfied by our logical clocks.

Let us identify  $\mathcal{S}$  with some set of "real" events in physical space-time, and let  $\rightarrow$  be the partial ordering of events defined by special relativity. One of the mysteries of the universe is that it is possible to construct a system of physical clocks which, running quite independently of one another, will satisfy the Strong Clock Condition. We can therefore use physical clocks to eliminate anomalous behavior. We now turn our attention to such clocks.

## Physical Clocks

Let us introduce a physical time coordinate into our space-time picture, and let  $C_i(t)$  denote the reading of the clock  $C_i$  at physical time  $t$ .<sup>8</sup> For mathematical con-

<sup>8</sup> We will assume a Newtonian space-time. If the relative motion of the clocks or gravitational effects are not negligible, then  $C_i(t)$  must be deduced from the actual clock reading by transforming from proper time to the arbitrarily chosen time coordinate.

venience, we assume that the clocks run continuously rather than in discrete "ticks." (A discrete clock can be thought of as a continuous one in which there is an error of up to  $\frac{1}{2}$  "tick" in reading it.) More precisely, we assume that  $C_i(t)$  is a continuous, differentiable function of  $t$  except for isolated jump discontinuities where the clock is reset. Then  $dC_i(t)/dt$  represents the rate at which the clock is running at time  $t$ .

In order for the clock  $C_i$  to be a true physical clock, it must run at approximately the correct rate. That is, we must have  $dC_i(t)/dt \approx 1$  for all  $t$ . More precisely, we will assume that the following condition is satisfied:

**PC1.** There exists a constant  $\kappa \ll 1$   
such that for all  $i$ :  $|dC_i(t)/dt - 1| < \kappa$ .

For typical crystal controlled clocks,  $\kappa \leq 10^{-6}$ .

It is not enough for the clocks individually to run at approximately the correct rate. They must be synchronized so that  $C_i(t) \approx C_j(t)$  for all  $i, j$ , and  $t$ . More precisely, there must be a sufficiently small constant  $\epsilon$  so that the following condition holds:

**PC2.** For all  $i, j$ :  $|C_i(t) - C_j(t)| < \epsilon$ .

If we consider vertical distance in Figure 2 to represent physical time, then PC2 states that the variation in height of a single tick line is less than  $\epsilon$ .

Since two different clocks will never run at exactly the same rate, they will tend to drift further and further apart. We must therefore devise an algorithm to insure that PC2 always holds. First, however, let us examine how small  $\kappa$  and  $\epsilon$  must be to prevent anomalous behavior. We must insure that the system  $\mathcal{L}$  of relevant physical events satisfies the Strong Clock Condition. We assume that our clocks satisfy the ordinary Clock Condition, so we need only require that the Strong Clock Condition holds when  $a$  and  $b$  are events in  $\mathcal{L}$  with  $a \rightarrow b$ . Hence, we need only consider events occurring in different processes.

Let  $\mu$  be a number such that if event  $a$  occurs at physical time  $t$  and event  $b$  in another process satisfies  $a \rightarrow b$ , then  $b$  occurs later than physical time  $t + \mu$ . In other words,  $\mu$  is less than the shortest transmission time for interprocess messages. We can always choose  $\mu$  equal to the shortest distance between processes divided by the speed of light. However, depending upon how messages in  $\mathcal{L}$  are transmitted,  $\mu$  could be significantly larger.

To avoid anomalous behavior, we must make sure that for any  $i, j$ , and  $t$ :  $C_i(t + \mu) - C_j(t) > 0$ . Combining this with PC1 and 2 allows us to relate the required smallness of  $\kappa$  and  $\epsilon$  to the value of  $\mu$  as follows. We assume that when a clock is reset, it is always set forward and never back. (Setting it back could cause C1 to be violated.) PC1 then implies that  $C_i(t + \mu) - C_i(t) > (1 - \kappa)\mu$ . Using PC2, it is then easy to deduce that  $C_i(t + \mu) - C_j(t) > 0$  if the following inequality holds:

$$\epsilon/(1 - \kappa) \leq \mu.$$

This inequality together with PC1 and PC2 implies that anomalous behavior is impossible.

We now describe our algorithm for insuring that PC2 holds. Let  $m$  be a message which is sent at physical time  $t$  and received at time  $t'$ . We define  $\nu_m = t' - t$  to be the *total delay* of the message  $m$ . This delay will, of course, not be known to the process which receives  $m$ . However, we assume that the receiving process knows some *minimum delay*  $\mu_m \geq 0$  such that  $\mu_m \leq \nu_m$ . We call  $\xi_m = \nu_m - \mu_m$  the *unpredictable delay* of the message.

We now specialize rules IR1 and 2 for our physical clocks as follows:

**IR1'.** For each  $i$ , if  $P_i$  does not receive a message at physical time  $t$ , then  $C_i$  is differentiable at  $t$  and  $dC_i(t)/dt > 0$ .

**IR2'.** (a) If  $P_i$  sends a message  $m$  at physical time  $t$ , then  $m$  contains a timestamp  $T_m = C_i(t)$ . (b) Upon receiving a message  $m$  at time  $t'$ , process  $P_j$  sets  $C_j(t')$  equal to maximum ( $C_j(t' - 0)$ ,  $T_m + \mu_m$ ).<sup>9</sup>

Although the rules are formally specified in terms of the physical time parameter, a process only needs to know its own clock reading and the timestamps of messages it receives. For mathematical convenience, we are assuming that each event occurs at a precise instant of physical time, and different events in the same process occur at different times. These rules are then specializations of rules IR1 and IR2, so our system of clocks satisfies the Clock Condition. The fact that real events have a finite duration causes no difficulty in implementing the algorithm. The only real concern in the implementation is making sure that the discrete clock ticks are frequent enough so C1 is maintained.

We now show that this clock synchronizing algorithm can be used to satisfy condition PC2. We assume that the system of processes is described by a directed graph in which an arc from process  $P_i$  to process  $P_j$  represents a communication line over which messages are sent directly from  $P_i$  to  $P_j$ . We say that a message is sent over this arc every  $\tau$  seconds if for any  $t$ ,  $P_i$  sends at least one message to  $P_j$  between physical times  $t$  and  $t + \tau$ . The *diameter* of the directed graph is the smallest number  $d$  such that for any pair of distinct processes  $P_j, P_k$ , there is a path from  $P_j$  to  $P_k$  having at most  $d$  arcs.

In addition to establishing PC2, the following theorem bounds the length of time it can take the clocks to become synchronized when the system is first started.

**THEOREM.** Assume a strongly connected graph of processes with diameter  $d$  which always obeys rules IR1' and IR2'. Assume that for any message  $m$ ,  $\mu_m \leq \mu$  for some constant  $\mu$ , and that for all  $t \geq t_0$ : (a) PC1 holds. (b) There are constants  $\tau$  and  $\xi$  such that every  $\tau$  seconds a message with an unpredictable delay less than  $\xi$  is sent over every arc. Then PC2 is satisfied with  $\epsilon \approx d(2\kappa\tau + \xi)$  for all  $t \geq t_0 + \tau d$ , where the approximations assume  $\mu + \xi \ll \tau$ .

The proof of this theorem is surprisingly difficult, and is given in the Appendix. There has been a great deal of work done on the problem of synchronizing physical clocks. We refer the reader to [4] for an intro-

<sup>9</sup>  $C_j(t' - 0) = \lim_{\delta \rightarrow 0} C_j(t' - |\delta|)$ .

duction to the subject. The methods described in the literature are useful for estimating the message delays  $\mu_m$  and for adjusting the clock frequencies  $dC_i/dt$  (for clocks which permit such an adjustment). However, the requirement that clocks are never set backwards seems to distinguish our situation from ones previously studied, and we believe this theorem to be a new result.

## Conclusion

We have seen that the concept of “happening before” defines an invariant partial ordering of the events in a distributed multiprocess system. We described an algorithm for extending that partial ordering to a somewhat arbitrary total ordering, and showed how this total ordering can be used to solve a simple synchronization problem. A future paper will show how this approach can be extended to solve any synchronization problem.

The total ordering defined by the algorithm is somewhat arbitrary. It can produce anomalous behavior if it disagrees with the ordering perceived by the system’s users. This can be prevented by the use of properly synchronized physical clocks. Our theorem showed how closely the clocks can be synchronized.

In a distributed system, it is important to realize that the order in which events occur is only a partial ordering. We believe that this idea is useful in understanding any multiprocess system. It should help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them.

## Appendix

### Proof of the Theorem

For any  $i$  and  $t$ , let us define  $C_i^t$  to be a clock which is set equal to  $C_i$  at time  $t$  and runs at the same rate as  $C_i$ , but is never reset. In other words,

$$C_i^t(t') = C_i(t) + \int_t^{t'} [dC_i(t)/dt] dt \quad (1)$$

for all  $t' \geq t$ . Note that

$$C_i(t') \geq C_i^t(t') \text{ for all } t' \geq t. \quad (2)$$

Suppose process  $P_1$  at time  $t_1$  sends a message to process  $P_2$  which is received at time  $t_2$  with an unpredictable delay  $\leq \xi$ , where  $t_0 \leq t_1 \leq t_2$ . Then for all  $t \geq t_2$  we have:

$$\begin{aligned} C_2^t(t) &\geq C_2^t(t_2) + (1 - \kappa)(t - t_2) && \text{[by (1) and PC1]} \\ &\geq C_1(t_1) + \mu_m + (1 - \kappa)(t - t_2) && \text{[by IR2' (b)]} \\ &= C_1(t_1) + (1 - \kappa)(t - t_1) - [(t_2 - t_1) - \mu_m] + \kappa(t_2 - t_1) \\ &\geq C_1(t_1) + (1 - \kappa)(t - t_1) - \xi. \end{aligned}$$

Hence, with these assumptions, for all  $t \geq t_2$  we have:

$$C_2^t(t) \geq C_1(t_1) + (1 - \kappa)(t - t_1) - \xi. \quad (3)$$

Now suppose that for  $i = 1, \dots, n$  we have  $t_i \leq t'_i <$

$t_{i+1}$ ,  $t_0 \leq t_1$ , and that at time  $t'_i$  process  $P_i$  sends a message to process  $P_{i+1}$  which is received at time  $t_{i+1}$  with an unpredictable delay less than  $\xi$ . Then repeated application of the inequality (3) yields the following result for  $t \geq t_{n+1}$ .

$$C_{n+1}^t(t) \geq C_1(t_1) + (1 - \kappa)(t - t_1) - n\xi. \quad (4)$$

From PC1, IR1' and 2' we deduce that

$$C_1(t_1) \geq C_1(t_1) + (1 - \kappa)(t_1' - t_1).$$

Combining this with (4) and using (2), we get

$$C_{n+1}(t) \geq C_1(t_1) + (1 - \kappa)(t - t_1) - n\xi \quad (5)$$

for  $t \geq t_{n+1}$ .

For any two processes  $P$  and  $P'$ , we can find a sequence of processes  $P = P_0, P_1, \dots, P_{n+1} = P'$ ,  $n \leq d$ , with communication arcs from each  $P_i$  to  $P_{i+1}$ . By hypothesis (b) we can find times  $t_i, t'_i$  with  $t'_i - t_i \leq \tau$  and  $t_{i+1} - t'_i \leq \nu$ , where  $\nu = \mu + \xi$ . Hence, an inequality of the form (5) holds with  $n \leq d$  whenever  $t \geq t_1 + d(\tau + \nu)$ . For any  $i, j$  and any  $t, t_1$  with  $t_1 \geq t_0$  and  $t \geq t_1 + d(\tau + \nu)$  we therefore have:

$$C_i(t) \geq C_j(t_1) + (1 - \kappa)(t - t_1) - d\xi. \quad (6)$$

Now let  $m$  be any message timestamped  $T_m$ , and suppose it is sent at time  $t$  and received at time  $t'$ . We pretend that  $m$  has a clock  $C_m$  which runs at a constant rate such that  $C_m(t) = t_m$  and  $C_m(t') = t_m + \mu_m$ . Then  $\mu_m \leq t' - t$  implies that  $dC_m/dt \leq 1$ . Rule IR2' (b) simply sets  $C_j(t')$  to maximum ( $C_j(t' - 0), C_m(t')$ ). Hence, clocks are reset only by setting them equal to other clocks.

For any time  $t_x \geq t_0 + \mu/(1 - \kappa)$ , let  $C_x$  be the clock having the largest value at time  $t_x$ . Since all clocks run at a rate less than  $1 + \kappa$ , we have for all  $i$  and all  $t \geq t_x$ :

$$C_i(t) \leq C_x(t_x) + (1 + \kappa)(t - t_x). \quad (7)$$

We now consider the following two cases: (i)  $C_x$  is the clock  $C_q$  of process  $P_q$ . (ii)  $C_x$  is the clock  $C_m$  of a message sent at time  $t_1$  by process  $P_q$ . In case (i), (7) simply becomes

$$C_i(t) \leq C_q(t_x) + (1 + \kappa)(t - t_x). \quad (8i)$$

In case (ii), since  $C_m(t_1) = C_q(t_1)$  and  $dC_m/dt \leq 1$ , we have

$$C_x(t_x) \leq C_q(t_1) + (t_x - t_1).$$

Hence, (7) yields

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \quad (8ii)$$

Since  $t_x \geq t_0 + \mu/(1 - \kappa)$ , we get

$$\begin{aligned} C_q(t_x) - \mu/(1 - \kappa) &\leq C_q(t_x) - \mu && \text{[by PC1]} \\ &\leq C_m(t_x) - \mu && \text{[by choice of } m \text{]} \\ &\leq C_m(t_x) - (t_x - t_1)\mu_m/\nu_m && \text{[} \mu_m \leq \mu, t_x - t_1 \leq \nu_m \text{]} \\ &= T_m && \text{[by definition of } C_m \text{]} \\ &= C_q(t_1) && \text{[by IR2'(a)].} \end{aligned}$$

Hence,  $C_q(t_x) - \mu/(1 - \kappa) \leq C_q(t_1)$ , so  $t_x - t_1 \leq \mu/(1 - \kappa)$  and thus  $t_1 \geq t_0$ .

Letting  $t_1 = t_x$  in case (i), we can combine (8i) and (8ii) to deduce that for any  $t, t_x$  with  $t \geq t_x \geq t_0 + \mu/(1 - \kappa)$  there is a process  $P_q$  and a time  $t_1$  with  $t_x - \mu/(1 - \kappa) \leq t_1 \leq t_x$  such that for all  $i$ :

$$C_i(t) \leq C_q(t_1) + (1 + \kappa)(t - t_1). \quad (9)$$

Choosing  $t$  and  $t_x$  with  $t \geq t_x + d(\tau + \nu)$ , we can combine (6) and (9) to conclude that there exists a  $t_1$  and a process  $P_q$  such that for all  $i$ :

$$\begin{aligned} C_q(t_1) + (1 - \kappa)(t - t_1) - d\xi &\leq C_i(t) \\ &\leq C_q(t_1) + (1 + \kappa)(t - t_1) \end{aligned} \quad (10)$$

Letting  $t = t_x + d(\tau + \nu)$ , we get

$$d(\tau + \nu) \leq t - t_1 \leq d(\tau + \nu) + \mu/(1 - \kappa).$$

Combining this with (10), we get

$$\begin{aligned} C_q(t_1) + (t - t_1) - \kappa d(\tau + \nu) - d\xi &\leq C_i(t) \leq C_q(t_1) \\ &\quad + (t - t_1) + \kappa[d(\tau + \nu) + \mu/(1 - \kappa)] \end{aligned} \quad (11)$$

Using the hypotheses that  $\kappa \ll 1$  and  $\mu \leq \nu \ll \tau$ , we can rewrite (11) as the following approximate inequality.

$$\begin{aligned} C_q(t_1) + (t - t_1) - d(\kappa\tau + \xi) &\leq C_i(t) \\ &\leq C_q(t_1) + (t - t_1) + d\kappa\tau. \end{aligned} \quad (12)$$

Since this holds for all  $i$ , we get

$$|C_i(t) - C_j(t)| \leq d(2\kappa\tau + \xi),$$

and this holds for all  $t \geq t_0 + d\tau$ .  $\square$

Note that relation (11) of the proof yields an exact upper bound for  $|C_i(t) - C_j(t)|$  in case the assumption  $\mu + \xi \ll \tau$  is invalid. An examination of the proof suggests a simple method for rapidly initializing the clocks, or resynchronizing them if they should go out of synchrony for any reason. Each process sends a message which is relayed to every other process. The procedure can be initiated by any process, and requires less than  $2d(\mu + \xi)$  seconds to effect the synchronization, assuming each of the messages has an unpredictable delay less than  $\xi$ .

*Acknowledgment.* The use of timestamps to order operations, and the concept of anomalous behavior are due to Paul Johnson and Robert Thomas.

Received March 1976; revised October 1977

#### References

1. Schwartz, J.T. *Relativity in Illustrations*. New York U. Press, New York, 1962.
2. Taylor, E.F., and Wheeler, J.A. *Space-Time Physics*, W.H. Freeman, San Francisco, 1966.
3. Lamport, L. The implementation of reliable distributed multiprocess systems. To appear in *Computer Networks*.
4. Ellingson, C., and Kulpinski, R.J. Dissemination of system-time. *IEEE Trans. Comm. Com-23*, 5 (May 1973), 605-624.

Programming  
Languages

J. J. Horning  
Editor

## Shallow Binding in Lisp 1.5

Henry G. Baker, Jr.  
Massachusetts Institute of Technology

Shallow binding is a scheme which allows the value of a variable to be accessed in a bounded amount of computation. An elegant model for shallow binding in Lisp 1.5 is presented in which context-switching is an environment tree transformation called rerooting. Rerooting is completely general and reversible, and is optional in the sense that a Lisp 1.5 interpreter will operate correctly whether or not rerooting is invoked on every context change. Since rerooting leaves `assoc [v, a]` invariant, for all variables  $v$  and all environments  $a$ , the programmer can have access to a rerooting primitive, `shallow[]`, which gives him dynamic control over whether accesses are shallow or deep, and which affects only the speed of execution of a program, not its semantics. In addition, multiple processes can be active in the same environment structure, so long as rerooting is an indivisible operation. Finally, the concept of rerooting is shown to combine the concept of shallow binding in Lisp with Dijkstra's display for Algol and hence is a general model for shallow binding.

**Key Words and Phrases:** Lisp 1.5, environment trees, FUNARG's, shallow binding, deep binding, multiprogramming, Algol display

**CR Categories:** 4.13, 4.22, 4.32

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0522.

Author's present address: Computer Science Department, University of Rochester, Rochester, NY 14627.

© 1978 ACM 0001-0782/78/0700-0565 \$00.75

Communications  
of  
the ACM

July 1978  
Volume 21  
Number 7

# Specifying Systems

First Printing

Version of 18 June 2002



# Specifying Systems

The TLA+ Language and Tools for  
Hardware and Software Engineers

Leslie Lamport

*Microsoft Research*

◆ Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal  
London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

U.S. Corporate and Government Sales  
(800) 382-3419  
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales  
(317) 581-3793  
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: [www.awprofessional.com](http://www.awprofessional.com)

*Library of Congress Cataloging-in-Publication Data*

Lamport, Leslie

Specifying systems : the TLA+ language and tools for hardware and software engineers / Leslie Lamport.

p. cm.

Includes bibliographical references and index.

ISBN 0-321-14306-X (alk. paper)

1. System design. 2. Computer systems--Specifications. 3. Logic, symbolic and mathematical. I. Title.

QA76.9.S88 L35 2003  
004.2'1--dc21

2002074369

Copyright © 2003 by Pearson Education, Inc

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.  
Rights and Contracts Department  
75 Arlington Street, Suite 300  
Boston, MA 02116  
Fax: (617) 848-7047

ISBN 0-321-14306-X  
Text printed on recycled paper  
1 2 3 4 5 6 7 8 9 10-MA-0605040302

*First printing, July 2002*

*To Ellen*



*This whole book is but a draught—nay, but the draught of a draught.*

Herman Melville



# Contents

|                                                     |             |
|-----------------------------------------------------|-------------|
| <b>List of Figures and Tables</b>                   | <b>xv</b>   |
| <b>Acknowledgments</b>                              | <b>xvii</b> |
| <b>Introduction</b>                                 | <b>1</b>    |
| <b>Part I Getting Started</b>                       | <b>5</b>    |
| <b>1 A Little Simple Math</b>                       | <b>9</b>    |
| 1.1 Propositional Logic . . . . .                   | 9           |
| 1.2 Sets . . . . .                                  | 11          |
| 1.3 Predicate Logic . . . . .                       | 12          |
| 1.4 Formulas and Language . . . . .                 | 14          |
| <b>2 Specifying a Simple Clock</b>                  | <b>15</b>   |
| 2.1 Behaviors . . . . .                             | 15          |
| 2.2 An Hour Clock . . . . .                         | 15          |
| 2.3 A Closer Look at the Specification . . . . .    | 18          |
| 2.4 The Specification in TLA <sup>+</sup> . . . . . | 19          |
| 2.5 An Alternative Specification . . . . .          | 21          |
| <b>3 An Asynchronous Interface</b>                  | <b>23</b>   |
| 3.1 The First Specification . . . . .               | 24          |
| 3.2 Another Specification . . . . .                 | 28          |
| 3.3 Types: A Reminder . . . . .                     | 30          |
| 3.4 Definitions . . . . .                           | 31          |
| 3.5 Comments . . . . .                              | 32          |
| <b>4 A FIFO</b>                                     | <b>35</b>   |
| 4.1 The Inner Specification . . . . .               | 35          |
| 4.2 Instantiation Examined . . . . .                | 37          |

|                |                                             |           |
|----------------|---------------------------------------------|-----------|
| 4.2.1          | Instantiation Is Substitution . . . . .     | 37        |
| 4.2.2          | Parametrized Instantiation . . . . .        | 39        |
| 4.2.3          | Implicit Substitutions . . . . .            | 40        |
| 4.2.4          | Instantiation Without Renaming . . . . .    | 40        |
| 4.3            | Hiding the Queue . . . . .                  | 41        |
| 4.4            | A Bounded FIFO . . . . .                    | 42        |
| 4.5            | What We're Specifying . . . . .             | 43        |
| <b>5</b>       | <b>A Caching Memory</b>                     | <b>45</b> |
| 5.1            | The Memory Interface . . . . .              | 45        |
| 5.2            | Functions . . . . .                         | 48        |
| 5.3            | A Linearizable Memory . . . . .             | 51        |
| 5.4            | Tuples as Functions . . . . .               | 53        |
| 5.5            | Recursive Function Definitions . . . . .    | 54        |
| 5.6            | A Write-Through Cache . . . . .             | 54        |
| 5.7            | Invariance . . . . .                        | 61        |
| 5.8            | Proving Implementation . . . . .            | 62        |
| <b>6</b>       | <b>Some More Math</b>                       | <b>65</b> |
| 6.1            | Sets . . . . .                              | 65        |
| 6.2            | Silly Expressions . . . . .                 | 67        |
| 6.3            | Recursion Revisited . . . . .               | 67        |
| 6.4            | Functions versus Operators . . . . .        | 69        |
| 6.5            | Using Functions . . . . .                   | 72        |
| 6.6            | Choose . . . . .                            | 73        |
| <b>7</b>       | <b>Writing a Specification: Some Advice</b> | <b>75</b> |
| 7.1            | Why Specify . . . . .                       | 75        |
| 7.2            | What to Specify . . . . .                   | 76        |
| 7.3            | The Grain of Atomicity . . . . .            | 76        |
| 7.4            | The Data Structures . . . . .               | 78        |
| 7.5            | Writing the Specification . . . . .         | 79        |
| 7.6            | Some Further Hints . . . . .                | 80        |
| 7.7            | When and How to Specify . . . . .           | 83        |
| <b>Part II</b> | <b>More Advanced Topics</b>                 | <b>85</b> |
| <b>8</b>       | <b>Liveness and Fairness</b>                | <b>87</b> |
| 8.1            | Temporal Formulas . . . . .                 | 88        |
| 8.2            | Temporal Tautologies . . . . .              | 92        |
| 8.3            | Temporal Proof Rules . . . . .              | 95        |
| 8.4            | Weak Fairness . . . . .                     | 96        |
| 8.5            | The Memory Specification . . . . .          | 100       |

|           |                                               |            |
|-----------|-----------------------------------------------|------------|
| 8.5.1     | The Liveness Requirement . . . . .            | 100        |
| 8.5.2     | Another Way to Write It . . . . .             | 101        |
| 8.5.3     | A Generalization . . . . .                    | 105        |
| 8.6       | Strong Fairness . . . . .                     | 106        |
| 8.7       | The Write-Through Cache . . . . .             | 107        |
| 8.8       | Quantification . . . . .                      | 109        |
| 8.9       | Temporal Logic Examined . . . . .             | 111        |
| 8.9.1     | A Review . . . . .                            | 111        |
| 8.9.2     | Machine Closure . . . . .                     | 111        |
| 8.9.3     | Machine Closure and Possibility . . . . .     | 113        |
| 8.9.4     | Refinement Mappings and Fairness . . . . .    | 114        |
| 8.9.5     | The Unimportance of Liveness . . . . .        | 116        |
| 8.9.6     | Temporal Logic Considered Confusing . . . . . | 116        |
| <b>9</b>  | <b>Real Time</b>                              | <b>117</b> |
| 9.1       | The Hour Clock Revisited . . . . .            | 117        |
| 9.2       | Real-Time Specifications in General . . . . . | 122        |
| 9.3       | A Real-Time Caching Memory . . . . .          | 124        |
| 9.4       | Zeno Specifications . . . . .                 | 128        |
| 9.5       | Hybrid System Specifications . . . . .        | 132        |
| 9.6       | Remarks on Real Time . . . . .                | 134        |
| <b>10</b> | <b>Composing Specifications</b>               | <b>135</b> |
| 10.1      | Composing Two Specifications . . . . .        | 136        |
| 10.2      | Composing Many Specifications . . . . .       | 138        |
| 10.3      | The FIFO . . . . .                            | 140        |
| 10.4      | Composition with Shared State . . . . .       | 142        |
| 10.4.1    | Explicit State Changes . . . . .              | 144        |
| 10.4.2    | Composition with Joint Actions . . . . .      | 147        |
| 10.5      | A Brief Review . . . . .                      | 150        |
| 10.5.1    | A Taxonomy of Composition . . . . .           | 151        |
| 10.5.2    | Interleaving Reconsidered . . . . .           | 151        |
| 10.5.3    | Joint Actions Reconsidered . . . . .          | 152        |
| 10.6      | Liveness and Hiding . . . . .                 | 152        |
| 10.6.1    | Liveness and Machine Closure . . . . .        | 152        |
| 10.6.2    | Hiding . . . . .                              | 154        |
| 10.7      | Open-System Specifications . . . . .          | 156        |
| 10.8      | Interface Refinement . . . . .                | 158        |
| 10.8.1    | A Binary Hour Clock . . . . .                 | 158        |
| 10.8.2    | Refining a Channel . . . . .                  | 159        |
| 10.8.3    | Interface Refinement in General . . . . .     | 163        |
| 10.8.4    | Open-System Specifications . . . . .          | 165        |
| 10.9      | Should You Compose? . . . . .                 | 167        |

|                                                       |            |
|-------------------------------------------------------|------------|
| <b>11 Advanced Examples</b>                           | <b>169</b> |
| 11.1 Specifying Data Structures . . . . .             | 170        |
| 11.1.1 Local Definitions . . . . .                    | 170        |
| 11.1.2 Graphs . . . . .                               | 172        |
| 11.1.3 Solving Differential Equations . . . . .       | 174        |
| 11.1.4 BNF Grammars . . . . .                         | 179        |
| 11.2 Other Memory Specifications . . . . .            | 183        |
| 11.2.1 The Interface . . . . .                        | 183        |
| 11.2.2 The Correctness Condition . . . . .            | 185        |
| 11.2.3 A Serial Memory . . . . .                      | 188        |
| 11.2.4 A Sequentially Consistent Memory . . . . .     | 195        |
| 11.2.5 The Memory Specifications Considered . . . . . | 200        |
| <b>Part III The Tools</b>                             | <b>205</b> |
| <b>12 The Syntactic Analyzer</b>                      | <b>207</b> |
| <b>13 The TLATEX Typesetter</b>                       | <b>211</b> |
| 13.1 Introduction . . . . .                           | 211        |
| 13.2 Comment Shading . . . . .                        | 212        |
| 13.3 How It Typesets the Specification . . . . .      | 213        |
| 13.4 How It Typesets Comments . . . . .               | 214        |
| 13.5 Adjusting the Output Format . . . . .            | 216        |
| 13.6 Output Files . . . . .                           | 217        |
| 13.7 Trouble-Shooting . . . . .                       | 218        |
| 13.8 Using LATEX Commands . . . . .                   | 219        |
| <b>14 The TLC Model Checker</b>                       | <b>221</b> |
| 14.1 Introduction to TLC . . . . .                    | 221        |
| 14.2 What TLC Can Cope With . . . . .                 | 230        |
| 14.2.1 TLC Values . . . . .                           | 230        |
| 14.2.2 How TLC Evaluates Expressions . . . . .        | 231        |
| 14.2.3 Assignment and Replacement . . . . .           | 234        |
| 14.2.4 Evaluating Temporal Formulas . . . . .         | 235        |
| 14.2.5 Overriding Modules . . . . .                   | 237        |
| 14.2.6 How TLC Computes States . . . . .              | 237        |
| 14.3 How TLC Checks Properties . . . . .              | 240        |
| 14.3.1 Model-Checking Mode . . . . .                  | 241        |
| 14.3.2 Simulation Mode . . . . .                      | 243        |
| 14.3.3 Views and Fingerprints . . . . .               | 243        |
| 14.3.4 Taking Advantage of Symmetry . . . . .         | 245        |
| 14.3.5 Limitations of Liveness Checking . . . . .     | 247        |
| 14.4 The <i>TLC</i> Module . . . . .                  | 248        |

|                                                        |     |
|--------------------------------------------------------|-----|
| 14.5 How to Use TLC . . . . .                          | 251 |
| 14.5.1 Running TLC . . . . .                           | 251 |
| 14.5.2 Debugging a Specification . . . . .             | 253 |
| 14.5.3 Hints on Using TLC Effectively . . . . .        | 257 |
| 14.6 What TLC Doesn't Do . . . . .                     | 262 |
| 14.7 The Fine Print . . . . .                          | 262 |
| 14.7.1 The Grammar of the Configuration File . . . . . | 262 |
| 14.7.2 Comparable TLC Values . . . . .                 | 264 |

## Part IV The TLA<sup>+</sup> Language 265

|                                                       |            |
|-------------------------------------------------------|------------|
| <i>Mini-Manual</i>                                    | 268–273    |
| <b>15 The Syntax of TLA<sup>+</sup></b>               | <b>275</b> |
| 15.1 The Simple Grammar . . . . .                     | 276        |
| 15.2 The Complete Grammar . . . . .                   | 283        |
| 15.2.1 Precedence and Associativity . . . . .         | 283        |
| 15.2.2 Alignment . . . . .                            | 286        |
| 15.2.3 Comments . . . . .                             | 288        |
| 15.2.4 Temporal Formulas . . . . .                    | 288        |
| 15.2.5 Two Anomalies . . . . .                        | 289        |
| 15.3 The Lexemes of TLA <sup>+</sup> . . . . .        | 289        |
| <b>16 The Operators of TLA<sup>+</sup></b>            | <b>291</b> |
| 16.1 Constant Operators . . . . .                     | 291        |
| 16.1.1 Boolean Operators . . . . .                    | 293        |
| 16.1.2 The Choose Operator . . . . .                  | 294        |
| 16.1.3 Interpretations of Boolean Operators . . . . . | 296        |
| 16.1.4 Conditional Constructs . . . . .               | 298        |
| 16.1.5 The Let/In Construct . . . . .                 | 299        |
| 16.1.6 The Operators of Set Theory . . . . .          | 299        |
| 16.1.7 Functions . . . . .                            | 301        |
| 16.1.8 Records . . . . .                              | 305        |
| 16.1.9 Tuples . . . . .                               | 306        |
| 16.1.10 Strings . . . . .                             | 307        |
| 16.1.11 Numbers . . . . .                             | 308        |
| 16.2 Nonconstant Operators . . . . .                  | 309        |
| 16.2.1 Basic Constant Expressions . . . . .           | 309        |
| 16.2.2 The Meaning of a State Function . . . . .      | 310        |
| 16.2.3 Action Operators . . . . .                     | 312        |
| 16.2.4 Temporal Operators . . . . .                   | 314        |

---

|                                                      |            |
|------------------------------------------------------|------------|
| <b>17 The Meaning of a Module</b>                    | <b>317</b> |
| 17.1 Operators and Expressions . . . . .             | 317        |
| 17.1.1 The Arity and Order of an Operator . . . . .  | 318        |
| 17.1.2 $\lambda$ Expressions . . . . .               | 319        |
| 17.1.3 Simplifying Operator Application . . . . .    | 320        |
| 17.1.4 Expressions . . . . .                         | 321        |
| 17.2 Levels . . . . .                                | 321        |
| 17.3 Contexts . . . . .                              | 324        |
| 17.4 The Meaning of a $\lambda$ Expression . . . . . | 325        |
| 17.5 The Meaning of a Module . . . . .               | 327        |
| 17.5.1 Extends . . . . .                             | 328        |
| 17.5.2 Declarations . . . . .                        | 329        |
| 17.5.3 Operator Definitions . . . . .                | 329        |
| 17.5.4 Function Definitions . . . . .                | 329        |
| 17.5.5 Instantiation . . . . .                       | 330        |
| 17.5.6 Theorems and Assumptions . . . . .            | 332        |
| 17.5.7 Submodules . . . . .                          | 332        |
| 17.6 Correctness of a Module . . . . .               | 332        |
| 17.7 Finding Modules . . . . .                       | 333        |
| 17.8 The Semantics of Instantiation . . . . .        | 334        |
| <b>18 The Standard Modules</b>                       | <b>339</b> |
| 18.1 Module <i>Sequences</i> . . . . .               | 339        |
| 18.2 Module <i>FiniteSets</i> . . . . .              | 340        |
| 18.3 Module <i>Bags</i> . . . . .                    | 340        |
| 18.4 The Numbers Modules . . . . .                   | 344        |

# List of Figures and Tables

## Figures

|      |                                                                               |     |
|------|-------------------------------------------------------------------------------|-----|
| 2.1  | The hour-clock specification—typeset and ASCII versions. . . . .              | 20  |
| 3.1  | Our first specification of an asynchronous interface. . . . .                 | 27  |
| 3.2  | Our second specification of an asynchronous interface. . . . .                | 30  |
| 3.3  | The hour-clock specification with comments. . . . .                           | 33  |
| 4.1  | The specification of a FIFO, with the internal variable $q$ visible. . . . .  | 38  |
| 4.2  | A specification of a FIFO buffer of length $N$ . . . . .                      | 43  |
| 5.1  | The specification of a memory interface. . . . .                              | 48  |
| 5.2  | The internal memory specification . . . . .                                   | 52  |
| 5.3  | The memory specification. . . . .                                             | 53  |
| 5.4  | The write-through cache. . . . .                                              | 55  |
| 5.5  | The write-through cache specification . . . . .                               | 57  |
| 9.1  | The real-time specification of an hour clock. . . . .                         | 121 |
| 9.2  | The <i>RealTime</i> module for writing real-time specifications. . . . .      | 125 |
| 9.3  | A real-time version of the linearizable memory specification. . . . .         | 126 |
| 9.4  | A real-time version of the write-through cache . . . . .                      | 129 |
| 10.1 | A noninterleaving composite specification of the FIFO. . . . .                | 143 |
| 10.2 | A joint-action specification of a linearizable memory. . . . .                | 150 |
| 10.3 | A specification of a binary hour clock. . . . .                               | 160 |
| 10.4 | Refining a channel. . . . .                                                   | 162 |
| 11.1 | A module for specifying operators on graphs. . . . .                          | 175 |
| 11.2 | A module for specifying the solution to a differential equation. .            | 178 |
| 11.3 | The definition of the grammar <i>GSE</i> for the language <i>SE</i> . . . . . | 183 |
| 11.4 | The module <i>BNFGrammars</i> . . . . .                                       | 184 |
| 11.5 | A module for specifying a register interface to a memory. . . . .             | 186 |
| 11.6 | Module <i>InnerSerial</i> . . . . .                                           | 196 |

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| 11.7 Module <i>InnerSequential</i> . . . . .                                 | 201 |
| 14.1 The alternating bit protocol . . . . .                                  | 223 |
| 14.2 Module <i>MCAlternatingBit</i> . . . . .                                | 227 |
| 14.3 A configuration file for module <i>MCAlternatingBit</i> . . . . .       | 227 |
| 14.4 A specification of correctness of the alternating bit protocol. . . . . | 229 |
| 14.5 The standard module <i>TLC</i> . . . . .                                | 248 |
| 14.6 The BNF grammar of the configuration file. . . . .                      | 263 |
| 18.1 The standard <i>Sequences</i> module. . . . .                           | 341 |
| 18.2 The standard <i>FiniteSets</i> module. . . . .                          | 341 |
| 18.3 The standard <i>Bags</i> module. . . . .                                | 343 |
| 18.4 The <i>Peano</i> module. . . . .                                        | 345 |
| 18.5 The <i>ProtoReals</i> module . . . . .                                  | 346 |
| 18.6 The standard <i>Naturals</i> module. . . . .                            | 348 |
| 18.7 The standard <i>Integers</i> module. . . . .                            | 348 |
| 18.8 The standard <i>Reals</i> module. . . . .                               | 348 |

## Tables

|                                                               |     |
|---------------------------------------------------------------|-----|
| Table 1 The constant operators. . . . .                       | 268 |
| Table 2 Miscellaneous constructs. . . . .                     | 269 |
| Table 3 Action operators. . . . .                             | 269 |
| Table 4 Temporal operators. . . . .                           | 269 |
| Table 5 User-definable operator symbols. . . . .              | 270 |
| Table 6 The precedence ranges of operators. . . . .           | 271 |
| Table 7 Operators defined in the standard modules. . . . .    | 272 |
| Table 8 The ASCII representations of typeset symbols. . . . . | 273 |

# Acknowledgments

I have spent more than two and a half decades learning how to specify and reason about concurrent computer systems. Before that, I had already spent many years learning how to use mathematics rigorously. I cannot begin to thank everyone who helped me during all that time. But I would like to express my gratitude to two men who, more than anyone else, influenced this book. Richard Palais taught me how even the most complicated mathematics could be made both rigorous and elegant. Martín Abadi influenced the development of TLA and was my collaborator in developing the ideas behind Chapters 9 and 10.

Much of what I know about applying the mathematics of TLA to the engineering problems of complex systems came from working with Mark Tuttle and Yuan Yu. Yuan Yu also helped turn TLA<sup>+</sup> into a useful tool for engineers by writing the TLC model checker, ignoring my warnings that it would never be practical. While writing the first version of the Syntactic Analyzer, Jean-Charles Grégoire helped me fine tune the TLA<sup>+</sup> language.

The following people made helpful comments on earlier versions of this book: Dominique Couturier, Douglas Frank, Vinod Grover, David Jefferson, Sara Kalvala, and Wolfgang Schreiner all pointed out mistakes. Kazuhiro Ogata read the manuscript with unusual care and found a number of mistakes. Kapila Pahalawatta found an error in the *ProtoReals* module. Paddy Krishnan also found an error in the *ProtoReals* module and suggested a way to improve the presentation. And I wish to extend my special thanks to Martin Rudalics, who read the manuscript with amazing thoroughness and caught many errors.

*Leslie Lamport  
Palo Alto, California  
4 March 2002*



# Introduction

This book will teach you how to write specifications of computer systems, using the language TLA<sup>+</sup>. It's rather long, but most people will read only Part I, which comprises the first 83 pages. That part contains all that most engineers need to know about writing specifications; it assumes only the basic background in computing and knowledge of mathematics expected of an undergraduate studying engineering or computer science. Part II contains more advanced material for more sophisticated readers. The remainder of the book is a reference manual—Part III for the TLA<sup>+</sup> tools and Part IV for the language itself.

The TLA World Wide Web page contains material to accompany the book, including the TLA<sup>+</sup> tools, exercises, references to the literature, and a list of corrections. There is a link to the TLA Web page on

<http://lamport.org>

You can also find the page by searching the Web for the 21-letter string

`uidlamporttlahomepage`

Do not put this string in any document that might appear on the Web.

## What Is a Specification?

*Writing is nature's way of letting you  
know how sloppy your thinking is.*  
— Guindon

A specification is a written description of what a system is supposed to do. Specifying a system helps us understand it. It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it.

This book is about specifying the behavioral properties of a system—also called its functional or logical properties. These are the properties that specify what the system is supposed to do. There are other important kinds of

properties that we don't consider, including performance properties. Worst-case performance can often be expressed as a behavioral property—for example, Chapter 9 explains how to specify that a system must react within a certain length of time. However, specifying average performance is beyond the scope of the methods described here.

Our basic tool for writing specifications is mathematics. Mathematics is nature's way of letting you know how sloppy your writing is. It's hard to be precise in an imprecise language like English or Chinese. In engineering, imprecision can lead to errors. To avoid errors, science and engineering have adopted mathematics as their language.

The mathematics we use is more formal than the math you've grown up with. Formal mathematics is nature's way of letting you know how sloppy your mathematics is. The mathematics written by most mathematicians and scientists is not really precise. It's precise in the small, but imprecise in the large. Each equation is a precise assertion, but you have to read the accompanying words to understand how the equations relate to one another and exactly what the theorems mean. Logicians have developed ways of eliminating those words and making the mathematics completely formal and, hence, completely precise.

Most mathematicians and scientists think that formal mathematics, without words, is long and tiresome. They're wrong. Ordinary mathematics can be expressed compactly in a precise, completely formal language. It takes only about two dozen lines to define the solution to an arbitrary differential equation in the *DifferentialEquations* module of Chapter 11. But few specifications need such sophisticated mathematics. Most require only simple application of a few standard mathematical concepts.

## Why TLA<sup>+</sup>?

We specify a system by describing its allowed behaviors—what it may do in the course of an execution. In 1977, Amir Pnueli introduced the use of temporal logic for describing system behaviors. In principle, a system could be described by a single temporal logic formula. In practice, it couldn't. Pnueli's temporal logic was ideal for describing some properties of systems, but awkward for others. So, it was usually combined with a more traditional way of describing systems.

In the late 1980's, I invented TLA, the Temporal Logic of Actions—a simple variant of Pnueli's original logic. TLA makes it practical to describe a system by a single formula. Most of a TLA specification consists of ordinary, nontemporal mathematics. Temporal logic plays a significant role only in describing those properties that it's good at describing. TLA also provides a nice way to formalize the style of reasoning about systems that has proved to be most effective in practice—a style known as *assertional* reasoning. However, this book is about specification; it says almost nothing about proofs.

---

Temporal logic assumes an underlying logic for expressing ordinary mathematics. There are many ways to formalize ordinary math. Most computer scientists prefer one that resembles their favorite programming language. I chose instead the one that most mathematicians prefer—the one logicians call first-order logic and set theory.

TLA provides a mathematical foundation for describing systems. To write specifications, we need a complete language built atop that foundation. I initially thought that this language should be some sort of abstract programming language whose semantics would be based on TLA. I didn't know what kind of programming language constructs would be best, so I decided to start writing specifications directly in TLA. I intended to introduce programming constructs as I needed them. To my surprise, I discovered that I didn't need them. What I needed was a robust language for writing mathematics.

Although mathematicians have developed the science of writing formulas, they haven't turned that science into an engineering discipline. They have developed notations for mathematics in the small, but not for mathematics in the large. The specification of a real system can be dozens or even hundreds of pages long. Mathematicians know how to write 20-line formulas, not 20-page formulas. So, I had to introduce notations for writing long formulas. What I took from programming languages were ideas for modularizing large specifications.

The language I came up with is called TLA<sup>+</sup>. I refined TLA<sup>+</sup> in the course of writing specifications of disparate systems. But it has changed little in the last few years. I have found TLA<sup>+</sup> to be quite good for specifying a wide class of systems—from program interfaces (APIs) to distributed systems. It can be used to write a precise, formal description of almost any sort of discrete system. It's especially well suited to describing asynchronous systems—that is, systems with components that do not operate in strict lock-step.

## About this Book

Part I, consisting of Chapters 1 through 7, is the core of the book and is meant to be read from beginning to end. It explains how to specify the class of properties known as *safety* properties. These properties, which can be specified with almost no temporal logic, are all that most engineers need to know about.

After reading Part I, you can read as much of Part II as you like. Each of its chapters is independent of the others. Temporal logic comes to the fore in Chapter 8, where it is used to specify the additional class of properties known as *liveness* properties. Chapter 9 describes how to specify real-time properties, and Chapter 10 describes how to write specifications as compositions. Chapter 11 contains more advanced examples.

Part III serves as the reference manual for three TLA<sup>+</sup> tools: the Syntactic Analyzer, the TLATEX typesetting program, and the TLC model checker. If

you want to use  $\text{TLA}^+$ , then you probably want to use these tools. They are available from the  $\text{TLA}$  Web page.  $\text{TLC}$  is the most sophisticated of them. The examples on the Web can get you started using it, but you'll have to read Chapter 14 to learn to use  $\text{TLC}$  effectively.

Part IV is a reference manual for the  $\text{TLA}^+$  language. Part I provides a good enough working knowledge of the language for most purposes. You need look at Part IV only if you have questions about the fine points of the syntax and semantics. Chapter 15 gives the syntax of  $\text{TLA}^+$ . Chapter 16 describes the precise meanings and the general forms of all the built-in operators of  $\text{TLA}^+$ ; Chapter 17 describes the precise meaning of all the higher-level  $\text{TLA}^+$  constructs such as definitions. Together, these two chapters specify the semantics of the language. Chapter 18 describes the standard modules—except for module *RealTime*, described in Chapter 9, and module *TLC*, described in Chapter 14. You might want to look at this chapter if you're curious about how standard elementary mathematics can be formalized in  $\text{TLA}^+$ .

Part IV does have something you may want to refer to often: a mini-manual that compactly presents lots of useful information. Pages 268–273 list all  $\text{TLA}^+$  operators, all user-definable symbols, the precedence of all operators, all operators defined in the standard modules, and the ASCII representation of symbols like  $\otimes$ .

# Part I

# Getting Started



---

A system specification consists of a lot of ordinary mathematics glued together with a tiny bit of temporal logic. That's why most TLA<sup>+</sup> constructs are for expressing ordinary mathematics. To write specifications, you have to be familiar with this ordinary math. Unfortunately, the computer science departments in many universities apparently believe that fluency in C++ is more important than a sound education in elementary mathematics. So, some readers may be unfamiliar with the math needed to write specifications. Fortunately, this math is quite simple. If exposure to C++ hasn't destroyed your ability to think logically, you should have no trouble filling any gaps in your mathematics education. You probably learned arithmetic before learning C++, so I will assume you know about numbers and arithmetic operations on them.<sup>1</sup> I will try to explain all other mathematical concepts that you need, starting in Chapter 1 with a review of some elementary math. I hope most readers will find this review completely unnecessary.

After the brief review of simple mathematics in the first chapter, Chapters 2 through 5 describe TLA<sup>+</sup> with a sequence of examples. Chapter 6 explains some more about the math used in writing specifications, and Chapter 7 reviews everything and provides some advice. By the time you finish Chapter 7, you should be able to handle most of the specification problems that you are likely to encounter in ordinary engineering practice.

---

<sup>1</sup>Some readers may need reminding that numbers are not strings of bits, and  $2^{33} * 2^{33}$  equals  $2^{66}$ , not *overflow error*.



# Chapter 1

## A Little Simple Math

### 1.1 Propositional Logic

Elementary algebra is the mathematics of real numbers and the operators  $+$ ,  $-$ ,  $*$  (multiplication), and  $/$  (division). Propositional logic is the mathematics of the two Boolean values TRUE and FALSE and the five operators whose names (and common pronunciations) are

|          |                   |               |                                |
|----------|-------------------|---------------|--------------------------------|
| $\wedge$ | conjunction (and) | $\Rightarrow$ | implication (implies)          |
| $\vee$   | disjunction (or)  | $\equiv$      | equivalence (is equivalent to) |
| $\neg$   | negation (not)    |               |                                |

To learn how to compute with numbers, you had to memorize addition and multiplication tables and algorithms for calculating with multidigit numbers. Propositional logic is much simpler, since there are only two values, TRUE and FALSE. To learn how to compute with these values, all you need to know are the following definitions of the five Boolean operators:

$\wedge$   $F \wedge G$  equals TRUE iff both  $F$  and  $G$  equal TRUE.

*iff* stands for *if and only if*. Like most mathematicians, I use *or* to mean *and/or*.

$\vee$   $F \vee G$  equals TRUE iff  $F$  or  $G$  equals TRUE (or both do).

$\neg$   $\neg F$  equals TRUE iff  $F$  equals FALSE.

$\Rightarrow$   $F \Rightarrow G$  equals TRUE iff  $F$  equals FALSE or  $G$  equals TRUE (or both).

$\equiv$   $F \equiv G$  equals TRUE iff  $F$  and  $G$  both equal TRUE or both equal FALSE.

We can also describe these operators by *truth tables*. This truth table gives the value of  $F \Rightarrow G$  for all four combinations of truth values of  $F$  and  $G$ :

| $F$   | $G$   | $F \Rightarrow G$ |
|-------|-------|-------------------|
| TRUE  | TRUE  | TRUE              |
| TRUE  | FALSE | FALSE             |
| FALSE | TRUE  | TRUE              |
| FALSE | FALSE | TRUE              |

The formula  $F \Rightarrow G$  asserts that  $F$  implies  $G$ —that is,  $F \Rightarrow G$  equals TRUE iff the statement “ $F$  implies  $G$ ” is true. People often find the definition of  $\Rightarrow$  confusing. They don’t understand why FALSE  $\Rightarrow$  TRUE and FALSE  $\Rightarrow$  FALSE should equal TRUE. The explanation is simple. We expect that if  $n$  is greater than 3, then it should be greater than 1, so  $n > 3$  should imply  $n > 1$ . Therefore, the formula  $(n > 3) \Rightarrow (n > 1)$  should equal TRUE. Substituting 4, 2, and 0 for  $n$  in this formula explains why  $F \Rightarrow G$  means  $F$  implies  $G$  or, equivalently, *if F then G*.

The equivalence operator  $\equiv$  is equality for Booleans. We can replace  $\equiv$  by  $=$ , but not vice versa. (We can write FALSE  $= \neg$ TRUE, but not  $2 + 2 \equiv 4$ .) It’s a good idea to write  $\equiv$  instead of  $=$  to make it clear that the equal expressions are Booleans.<sup>1</sup>

Just like formulas of algebra, formulas of propositional logic are made up of values, operators, and identifiers like  $x$  that stand for values. However, propositional-logic formulas use only the two values TRUE and FALSE and the five Boolean operators  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\Rightarrow$ , and  $\equiv$ . In algebraic formulas,  $*$  has higher precedence (binds more tightly) than  $+$ , so  $x+y*z$  means  $x+(y*z)$ . Similarly,  $\neg$  has higher precedence than  $\wedge$  and  $\vee$ , which have higher precedence than  $\Rightarrow$  and  $\equiv$ , so  $\neg F \wedge G \Rightarrow H$  means  $((\neg F) \wedge G) \Rightarrow H$ . Other mathematical operators like  $+$  and  $>$  have higher precedence than the operators of propositional logic, so  $n > 0 \Rightarrow n - 1 \geq 0$  means  $(n > 0) \Rightarrow (n - 1 \geq 0)$ . Redundant parentheses can’t hurt and often make a formula easier to read. If you have the slightest doubt about whether parentheses are needed, use them.

The operators  $\wedge$  and  $\vee$  are associative, just like  $+$  and  $*$ . Associativity of  $+$  means that  $x + (y + z)$  equals  $(x + y) + z$ , so we can write  $x + y + z$  without parentheses. Similarly, associativity of  $\wedge$  and  $\vee$  lets us write  $F \wedge G \wedge H$  or  $F \vee G \vee H$ . Like  $+$  and  $*$ , the operators  $\wedge$  and  $\vee$  are also commutative, so  $F \wedge G$  is equivalent to  $G \wedge F$ , and  $F \vee G$  is equivalent to  $G \vee F$ .

The truth of the formula  $(x = 2) \Rightarrow (x + 1 = 3)$  expresses a fact about numbers. To determine that it’s true, we have to understand some elementary properties of arithmetic. However, we can tell that  $(x = 2) \Rightarrow (x = 2) \vee (y > 7)$  is true even if we know nothing about numbers. This formula is true because  $F \Rightarrow F \vee G$  is true, regardless of what the formulas  $F$  and  $G$  are. In other

<sup>1</sup>Section 16.1.3 on page 296 explains a more subtle reason for using  $\equiv$  instead of  $=$  for equality of Boolean values.

words,  $F \Rightarrow F \vee G$  is true for all possible truth values of its identifiers  $F$  and  $G$ . Such a formula is called a *tautology*.

In general, a tautology of propositional logic is a propositional-logic formula that is true for all possible truth values of its identifiers. Simple tautologies like this should be as obvious as simple algebraic properties of numbers. It should be as obvious that  $F \Rightarrow F \vee G$  is a tautology as that  $x \leq x + y$  is true for all non-negative numbers  $x$  and  $y$ . One can derive complicated tautologies from simpler ones by calculations, just as one derives more complicated properties of numbers from simpler ones. However, this takes practice. You've spent years learning how to manipulate number-valued expressions—for example, to deduce that  $x \leq -x + y$  holds iff  $2*x \leq y$  does. You probably haven't learned to deduce that  $\neg F \vee G$  holds iff  $F \Rightarrow G$  does.

If you haven't learned to manipulate Boolean-valued expressions, you will have to do the equivalent of counting on your fingers. You can check if a formula is a tautology by calculating whether it equals TRUE for each possible assignment of Boolean values to its variables. This is best done by constructing a truth table that lists the possible assignments of values to variables and the corresponding values of all subformulas. For example, here is the truth table showing that  $(F \Rightarrow G) \equiv (\neg F \vee G)$  is a tautology.

| $F$   | $G$   | $F \Rightarrow G$ | $\neg F$ | $\neg F \vee G$ | $(F \Rightarrow G) \equiv \neg F \vee G$ |
|-------|-------|-------------------|----------|-----------------|------------------------------------------|
| TRUE  | TRUE  | TRUE              | FALSE    | TRUE            | TRUE                                     |
| TRUE  | FALSE | FALSE             | FALSE    | FALSE           | TRUE                                     |
| FALSE | TRUE  | TRUE              | TRUE     | TRUE            | TRUE                                     |
| FALSE | FALSE | TRUE              | TRUE     | TRUE            | TRUE                                     |

Writing truth tables is a good way to improve your understanding of propositional logic. However, computers are better than people at doing this sort of calculation. Chapter 14 explains, on page 261, how to use the TLC model checker to verify propositional logic tautologies and to perform other TLA<sup>+</sup> calculations.

## 1.2 Sets

Set theory is the foundation of ordinary mathematics. A set is often described as a collection of elements, but saying that a set is a collection doesn't explain very much. The concept of set is so fundamental that we don't try to define it. We take as undefined concepts the notion of a set and the relation  $\in$ , where  $x \in S$  means that  $x$  is an element of  $S$ . We often say *is in* instead of *is an element of*.

A set can have a finite or infinite number of elements. The set of all natural numbers (0, 1, 2, etc.) is an infinite set. The set of all natural numbers less than

3 is finite, and contains the three elements 0, 1, and 2. We can write this set  $\{0, 1, 2\}$ .

A set is completely determined by its elements. Two sets are equal iff they have the same elements. Thus,  $\{0, 1, 2\}$  and  $\{2, 1, 0\}$  and  $\{0, 0, 1, 2, 2\}$  are all the same set—the unique set containing the three elements 0, 1, and 2. The empty set, which we write  $\{\}$ , is the unique set that has no elements.

The most common operations on sets are

$$\cap \text{ intersection} \quad \cup \text{ union} \quad \subseteq \text{ subset} \quad \setminus \text{ set difference}$$

Here are their definitions and examples of their use.

$S \cap T$  The set of elements in both  $S$  and  $T$ .

$$\{1, -1/2, 3\} \cap \{1, 2, 3, 5, 7\} = \{1, 3\}$$

$S \cup T$  The set of elements in  $S$  or  $T$  (or both).

$$\{1, -1/2\} \cup \{1, 5, 7\} = \{1, -1/2, 5, 7\}$$

$S \subseteq T$  True iff every element of  $S$  is an element of  $T$ .

$$\{1, 3\} \subseteq \{3, 2, 1\}$$

$S \setminus T$  The set of elements in  $S$  that are not in  $T$ .

$$\{1, -1/2, 3\} \setminus \{1, 5, 7\} = \{-1/2, 3\}$$

This is all you need to know about sets before we start looking at how to specify systems. We'll return to set theory in Section 6.1.

## 1.3 Predicate Logic

Once we have sets, it's natural to say that some formula is true for all the elements of a set, or for some of the elements of a set. Predicate logic extends propositional logic with the two quantifiers

$\forall$  universal quantification (for all)

$\exists$  existential quantification (there exists)

The formula  $\forall x \in S : F$  asserts that formula  $F$  is true for every element  $x$  in the set  $S$ . For example,  $\forall n \in \text{Nat} : n + 1 > n$  asserts that the formula  $n + 1 > n$  is true for all elements  $n$  of the set  $\text{Nat}$  of natural numbers. This formula happens to be true.

The formula  $\exists x \in S : F$  asserts that formula  $F$  is true for at least one element  $x$  in  $S$ . For example,  $\exists n \in \text{Nat} : n^2 = 2$  asserts that there exists a natural number  $n$  whose square equals 2. This formula happens to be false.

Formula  $F$  is true for some  $x$  in  $S$  iff  $F$  is not false for all  $x$  in  $S$ —that is, iff it's not the case that  $\neg F$  is true for all  $x$  in  $S$ . Hence, the formula

$$(1.1) \quad (\exists x \in S : F) \equiv \neg(\forall x \in S : \neg F)$$

is a tautology of predicate logic, meaning that it is true for all values of the identifiers  $S$  and  $F$ .<sup>2</sup>

Since there exists no element in the empty set, the formula  $\exists x \in \{\} : F$  is false for every formula  $F$ . By (1.1), this implies that  $\forall x \in \{\} : F$  must be true for every  $F$ .

The quantification in the formulas  $\forall x \in S : F$  and  $\exists x \in S : F$  is said to be *bounded*, since these formulas make an assertion only about elements in the set  $S$ . There is also unbounded quantification. The formula  $\forall x : F$  asserts that  $F$  is true for all values  $x$ , and  $\exists x : F$  asserts that  $F$  is true for at least one value of  $x$ —a value that is not constrained to be in any particular set. Bounded and unbounded quantification are related by the following tautologies:

$$\begin{aligned} (\forall x \in S : F) &\equiv (\forall x : (x \in S) \Rightarrow F) \\ (\exists x \in S : F) &\equiv (\exists x : (x \in S) \wedge F) \end{aligned}$$

The analog of (1.1) for unbounded quantifiers is also a tautology:

$$(\exists x : F) \equiv \neg(\forall x : \neg F)$$

Whenever possible, it is better to use bounded than unbounded quantification in a specification. This makes the specification easier for both people and tools to understand.

Universal quantification generalizes conjunction. If  $S$  is a finite set, then  $\forall x \in S : F$  is the conjunction of the formulas obtained by substituting the different elements of  $S$  for  $x$  in  $F$ . For example,

$$(\forall x \in \{2, 3, 7\} : x < y^x) \equiv (2 < y^2) \wedge (3 < y^3) \wedge (7 < y^7)$$

We sometimes informally talk about the conjunction of an infinite number of formulas when we formally mean a universally quantified formula. For example, the conjunction of the formulas  $x \leq y^x$  for all natural numbers  $x$  is the formula  $\forall x \in \text{Nat} : x \leq y^x$ . Similarly, existential quantification generalizes disjunction.

Logicians have rules for proving predicate-logic tautologies such as (1.1), but you shouldn't need them. You should become familiar enough with predicate logic that simple tautologies are obvious. Thinking of  $\forall$  as conjunction and  $\exists$  as disjunction can help. For example, the associativity and commutativity of conjunction and disjunction lead to the tautologies

$$\begin{aligned} (\forall x \in S : F) \wedge (\forall x \in S : G) &\equiv (\forall x \in S : F \wedge G) \\ (\exists x \in S : F) \vee (\exists x \in S : G) &\equiv (\exists x \in S : F \vee G) \end{aligned}$$

for any set  $S$  and formulas  $F$  and  $G$ .

Mathematicians use some obvious abbreviations for nested quantifiers. For example,

---

<sup>2</sup>Strictly speaking,  $\in$  isn't an operator of predicate logic, so this isn't really a predicate-logic tautology.

$$\begin{aligned}\forall x \in S, y \in T : F &\text{ means } \forall x \in S : (\forall y \in T : F) \\ \exists w, x, y, z \in S : F &\text{ means } \exists w \in S : (\exists x \in S : (\exists y \in S : (\exists z \in S : F)))\end{aligned}$$

In the expression  $\exists x \in S : F$ , logicians say that  $x$  is a *bound variable* and that occurrences of  $x$  in  $F$  are *bound*. For example,  $n$  is a bound variable in the formula  $\exists n \in \text{Nat} : n + 1 > n$ , and the two occurrences of  $n$  in the subexpression  $n + 1 > n$  are bound. A variable  $x$  that's not bound is said to be *free*, and occurrences of  $x$  that are not bound are called *free* occurrences. This terminology is rather misleading. A bound variable doesn't really occur in a formula because replacing it by some new variable doesn't change the formula. The two formulas

$$\exists n \in \text{Nat} : n + 1 > n \qquad \exists x \in \text{Nat} : x + 1 > x$$

are equivalent. Calling  $n$  a variable of the first formula is a bit like calling  $a$  a variable of that formula because it appears in the name *Nat*. Nevertheless, it is convenient to talk about an occurrence of a bound variable in a formula.

## 1.4 Formulas and Language

When you first studied mathematics, formulas were statements. The formula  $2 * x > x$  was just a compact way of writing the statement “2 times  $x$  is greater than  $x$ .” In this book, you are entering the realm of logic, where a formula is a noun. The formula  $2 * x > x$  is just a formula; it may be true or false, depending on the value of  $x$ . If we want to assert that this formula is true, meaning that  $2 * x$  really is greater than  $x$ , we should explicitly write “ $2 * x > x$  is true.”

Using a formula in place of a statement can lead to confusion. On the other hand, formulas are more compact and easier to read than prose. Reading  $2 * x > x$  is easier than reading “ $2 * x$  is greater than  $x$ ”; and “ $2 * x > x$  is true” may seem redundant. So, like most mathematicians, I will often write sentences like

We know that  $x$  is positive, so  $2 * x > x$ .

If it's not obvious whether a formula is really a formula or is the statement that the formula is true, here's an easy way to tell. Replace the formula with a name and read the sentence. If the sentence is grammatically correct, even though nonsensical, then the formula is a formula; otherwise, it's a statement. The formula  $2 * x > x$  in the sentence above is a statement because

We know that  $x$  is positive, so Mary.

is ungrammatical. It is a formula in the sentence

To prove  $2 * x > x$ , we must prove that  $x$  is positive.

because the following silly sentence is grammatically correct:

To prove Fred, we must prove that  $x$  is positive.

# Chapter 2

## Specifying a Simple Clock

### 2.1 Behaviors

Before we try to specify a system, let's look at how scientists do it. For centuries, they have described a system with equations that determine how its state evolves with time, where the state consists of the values of variables. For example, the state of the system comprising the earth and the moon might be described by the values of the four variables  $e\_pos$ ,  $m\_pos$ ,  $e\_vel$ , and  $m\_vel$ , representing the positions and velocities of the two bodies. These values are elements in a 3-dimensional space. The earth-moon system is described by equations expressing the variables' values as functions of time and of certain constants—namely, their masses and initial positions and velocities.

A behavior of the earth-moon system consists of a function  $F$  from time to states,  $F(t)$  representing the state of the system at time  $t$ . A computer system differs from the systems traditionally studied by scientists because we can pretend that its state changes in discrete steps. So, we represent the execution of a system as a sequence of states. Formally, we define a *behavior* to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system.

### 2.2 An Hour Clock

Let's start with a very trivial system—a digital clock that displays only the hour. To make the system completely trivial, we ignore the relation between the display and the actual time. The hour clock is then just a device whose display cycles through the values 1 through 12. Let the variable  $hr$  represent the clock's

display. A typical behavior of the clock is the sequence

$$(2.1) \quad [hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 1] \rightarrow [hr = 2] \rightarrow \dots$$

of states, where  $[hr = 11]$  is a state in which the variable  $hr$  has the value 11. A pair of successive states, such as  $[hr = 1] \rightarrow [hr = 2]$ , is called a *step*.

To specify the hour clock, we describe all its possible behaviors. We write an *initial predicate* that specifies the possible initial values of  $hr$ , and a *next-state relation* that specifies how the value of  $hr$  can change in any step.

We don't want to specify exactly what the display reads initially; any hour will do. So, we want the initial predicate to assert that  $hr$  can have any value from 1 through 12. Let's call the initial predicate  $HCini$ . We might informally define  $HCini$  by

$$HCini \triangleq hr \in \{1, \dots, 12\}$$

The symbol  $\triangleq$   
means *is defined  
to equal*.

Later, we'll see how to write this definition formally, without the “ $\dots$ ” that stands for the informal *and so on*.

The next-state relation  $HCnxt$  is a formula expressing the relation between the values of  $hr$  in the old (first) state and new (second) state of a step. We let  $hr$  represent the value of  $hr$  in the old state and  $hr'$  represent its value in the new state. (The ' $'$  in  $hr'$  is read *prime*.) We want the next-state relation to assert that  $hr'$  equals  $hr + 1$  except if  $hr$  equals 12, in which case  $hr'$  should equal 1. Using an IF/THEN/ELSE construct with the obvious meaning, we can define  $HCnxt$  to be the next-state relation by writing

$$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

$HCnxt$  is an ordinary mathematical formula, except that it contains primed as well as unprimed variables. Such a formula is called an *action*. An action is true or false of a step. A step that satisfies the action  $HCnxt$  is called an  $HCnxt$  *step*.

When an  $HCnxt$  step occurs, we sometimes say that  $HCnxt$  is *executed*. However, it would be a mistake to take this terminology seriously. An action is a formula, and formulas aren't executed.

We want our specification to be a single formula, not the pair of formulas  $HCini$  and  $HCnxt$ . This formula must assert about a behavior that (i) its initial state satisfies  $HCini$ , and (ii) each of its steps satisfies  $HCnxt$ . We express (i) as the formula  $HCini$ , which we interpret as a statement about behaviors to mean that the initial state satisfies  $HCini$ . To express (ii), we use the temporal-logic operator  $\square$  (pronounced *box*). The temporal formula  $\square F$  asserts that formula  $F$  is always true. In particular,  $\square HCnxt$  is the assertion that  $HCnxt$  is true for every step in the behavior. So,  $HCini \wedge \square HCnxt$  is true of a behavior iff the initial state satisfies  $HCini$  and every step satisfies  $HCnxt$ . This formula describes all behaviors like the one in (2.1) on this page; it seems to be the specification we're looking for.

If we considered the clock only in isolation and never tried to relate it to another system, then this would be a fine specification. However, suppose the clock is part of a larger system—for example, the hour display of a weather station that displays the current hour and temperature. The state of the station is described by two variables:  $hr$ , representing the hour display, and  $tmp$ , representing the temperature display. Consider this behavior of the weather station:

$$\begin{bmatrix} hr = 11 \\ tmp = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 12 \\ tmp = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 12 \\ tmp = 23.4 \end{bmatrix} \rightarrow \\ \begin{bmatrix} hr = 12 \\ tmp = 23.3 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 1 \\ tmp = 23.3 \end{bmatrix} \rightarrow \dots$$

In the second and third steps,  $tmp$  changes but  $hr$  remains the same. These steps are not allowed by  $\square HC_{nxt}$ , which asserts that every step must increment  $hr$ . The formula  $HC_{ini} \wedge \square HC_{nxt}$  does not describe the hour clock in the weather station.

A formula that describes *any* hour clock must allow steps that leave  $hr$  unchanged—in other words,  $hr' = hr$  steps. These are called *stuttering steps* of the clock. A specification of the hour clock should allow both  $HC_{nxt}$  steps and stuttering steps. So, a step should be allowed iff it is either an  $HC_{nxt}$  step or a stuttering step—that is, iff it is a step satisfying  $HC_{nxt} \vee (hr' = hr)$ . This suggests that we adopt  $HC_{ini} \wedge \square(HC_{nxt} \vee (hr' = hr))$  as our specification. In TLA, we let  $[HC_{nxt}]_{hr}$  stand for  $HC_{nxt} \vee (hr' = hr)$ , so we can write the formula more compactly as  $HC_{ini} \wedge \square[HC_{nxt}]_{hr}$ .

The formula  $HC_{ini} \wedge \square[HC_{nxt}]_{hr}$  does allow stuttering steps. In fact, it allows the behavior

$$[hr = 10] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow \dots$$

that ends with an infinite sequence of stuttering steps. This behavior describes a clock whose display attains the value 11 and then keeps that value forever—in other words, a clock that stops at 11. In a like manner, we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones.

It's natural to require that a clock does not stop, so our specification should assert that there are infinitely many nonstuttering steps. Chapter 8 explains how to express this requirement. For now, we content ourselves with clocks that may stop, and we take as our specification of an hour clock the formula  $HC$  defined by

$$HC \triangleq HC_{ini} \wedge \square[HC_{nxt}]_{hr}$$

I pronounce  $[HC_{nxt}]_{hr}$  as square  $HC_{nxt}$  sub  $hr$ .

## 2.3 A Closer Look at the Specification

A state is an assignment of values to variables, but what variables? The answer is simple: all variables. In the behavior (2.1) on page 16,  $[hr = 1]$  represents some particular state that assigns the value 1 to  $hr$ . It might assign the value 23 to the variable  $tmp$  and the value  $\sqrt{-17}$  to the variable  $m\_pos$ . We can think of a state as representing a potential state of the entire universe. A state that assigns 1 to  $hr$  and a particular point in 3-space to  $m\_pos$  describes a state of the universe in which the hour clock reads 1 and the moon is in a particular place. A state that assigns  $\sqrt{-2}$  to  $hr$  doesn't correspond to any state of the universe that we recognize, because the hour clock can't display the value  $\sqrt{-2}$ . It might represent the state of the universe after a bomb fell on the clock, making its display purely imaginary.

A behavior is an infinite sequence of states—for example:

$$(2.2) \quad [hr = 11] \rightarrow [hr = 77.2] \rightarrow [hr = 78.2] \rightarrow [hr = \sqrt{-2}] \rightarrow \dots$$

A behavior describes a potential history of the universe. The behavior (2.2) doesn't correspond to a history that we understand, because we don't know how the clock's display can change from 11 to 77.2. Whatever kind of history it represents is not one in which the clock is doing what it's supposed to.

Formula  $HC$  is a temporal formula. A temporal formula is an assertion about behaviors. We say that a behavior *satisfies*  $HC$  iff  $HC$  is a true assertion about the behavior. Behavior (2.1) satisfies formula  $HC$ . Behavior (2.2) does not, because  $HC$  asserts that every step satisfies  $HC_{nxt}$  or leaves  $hr$  unchanged, and the first and third steps of (2.2) don't. (The second step,  $[hr = 77.2] \rightarrow [hr = 78.2]$ , does satisfy  $HC_{nxt}$ .) We regard formula  $HC$  to be the specification of an hour clock because it is satisfied by exactly those behaviors that represent histories of the universe in which the clock functions properly.

If the clock is behaving properly, then its display should be an integer from 1 through 12. So,  $hr$  should be an integer from 1 through 12 in every state of any behavior satisfying the clock's specification,  $HC$ . Formula  $HC_{ini}$  asserts that  $hr$  is an integer from 1 through 12, and  $\square HC_{ini}$  asserts that  $HC_{ini}$  is always true. So,  $\square HC_{ini}$  should be true for any behavior satisfying  $HC$ . Another way of saying this is that  $HC$  implies  $\square HC_{ini}$ , for any behavior. Thus, the formula  $HC \Rightarrow \square HC_{ini}$  should be satisfied by *every* behavior. A temporal formula satisfied by every behavior is called a *theorem*, so  $HC \Rightarrow \square HC_{ini}$  should be a theorem.<sup>1</sup> It's easy to see that it is:  $HC$  implies that  $HC_{ini}$  is true initially (in the first state of the behavior), and  $\square [HC_{nxt}]_{hr}$  implies that each step either advances  $hr$  to its proper next value or else leaves  $hr$  unchanged. We can formalize this reasoning using the proof rules of TLA, but we're not going to delve into proofs and proof rules.

---

<sup>1</sup>Logicians call a formula *valid* if it is satisfied by every behavior; they reserve the term *theorem* for provably valid formulas.

## 2.4 The Specification in TLA<sup>+</sup>

Figure 2.1 on the next page shows how the hour-clock specification can be written in TLA<sup>+</sup>. There are two versions: the ASCII version on the bottom is the actual TLA<sup>+</sup> specification, the way you type it; the typeset version on the top is one that the TLATEX program, described in Chapter 13, might produce. Before trying to understand the specification, observe the relation between the two syntaxes.

- Reserved words that appear in small upper-case letters (like EXTENDS) are written in ASCII with ordinary upper-case letters.
- When possible, symbols are represented pictorially in ASCII—for example,  $\square$  is typed as `[]` and  $\neq$  as `#`. (You can also type  $\neq$  as `/=`.)
- When there is no good ASCII representation, T<sub>E</sub>X notation<sup>2</sup> is used—for example,  $\in$  is typed as `\in`. The major exception is  $\triangleq$ , which is typed as `==`.

A complete list of symbols and their ASCII equivalents appears in Table 8 on page 273. I will usually show the typeset version of a specification; the ASCII versions of all the specifications in this book can be found through the TLA Web page.

Now let's look at what the specification says. It starts with

---

MODULE *HourClock*

---

which begins a module named *HourClock*. TLA<sup>+</sup> specifications are partitioned into modules; the hour clock's specification consists of this single module.

Arithmetic operators like  $+$  are not built into TLA<sup>+</sup>, but are themselves defined in modules. (You might want to write a specification in which  $+$  means addition of matrices rather than numbers.) The usual operators on natural numbers are defined in the *Naturals* module. Their definitions are incorporated into module *HourClock* by the statement

EXTENDS *Naturals*

Every symbol that appears in a formula must either be a built-in operator of TLA<sup>+</sup>, or else it must be declared or defined. The statement

VARIABLE *hr*

declares *hr* to be a variable.

---

<sup>2</sup>The T<sub>E</sub>X typesetting system is described in *The T<sub>E</sub>Xbook* by Donald E. Knuth, published by Addison-Wesley, Reading, Massachusetts, 1986.

---

 MODULE *HourClock*


---

```

EXTENDS Naturals
VARIABLE hr
HCini \triangleq hr \in (1 .. 12)
HCnxt \triangleq hr' = IF hr \neq 12 THEN hr + 1 ELSE 1
HC \triangleq HCini \wedge \square [HCnxt]hr

```

---

THEOREM  $HC \Rightarrow \square HCini$

---



---

 MODULE *HourClock*


---

```

EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [] [HCnxt]_hr

```

---

THEOREM  $HC \Rightarrow [] HCini$

---

**Figure 2.1:** The hour-clock specification—typeset and ASCII versions.

To define  $HCini$ , we need to express the set  $\{1, \dots, 12\}$  formally, without the ellipsis “ $\dots$ ”. We can write this set out completely as

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

but that’s tiresome. Instead, we use the operator “ $\dots$ ”, defined in the *Naturals* module, to write this set as  $1..12$ . In general  $i..j$  is the set of integers from  $i$  through  $j$ , for any integers  $i$  and  $j$ . (It equals the empty set if  $j < i$ .) It’s now obvious how to write the definition of  $HCini$ . The definitions of  $HCnxt$  and  $HC$  are written just as before. (The ordinary mathematical operators of logic and set theory, like  $\wedge$  and  $\in$ , are built into TLA<sup>+</sup>.)

The line

---

can appear anywhere between statements; it’s purely cosmetic and has no meaning. Following it is the statement

THEOREM  $HC \Rightarrow \square HCini$

of the theorem that was discussed above. This statement asserts that the formula  $HC \Rightarrow \square HCini$  is true in the context of the statement. More precisely, it

asserts that the formula follows logically from the definitions in this module, the definitions in the *Naturals* module, and the rules of TLA<sup>+</sup>. If the formula were not true, then the module would be incorrect.

The module is terminated by the symbol

---

The specification of the hour clock is the definition of  $HC$ , including the definitions of the formulas  $HC_{nxt}$  and  $HC_{ini}$  and of the operators  $\dots$  and  $+$  that appear in the definition of  $HC$ . Formally, nothing in the module tells us that  $HC$  rather than  $HC_{ini}$  is the clock's specification. TLA<sup>+</sup> is a language for writing mathematics—in particular, for writing mathematical definitions and theorems. What those definitions represent, and what significance we attach to those theorems, lies outside the scope of mathematics and therefore outside the scope of TLA<sup>+</sup>. Engineering requires not just the ability to use mathematics, but the ability to understand what, if anything, the mathematics tells us about an actual system.

## 2.5 An Alternative Specification

The *Naturals* module also defines the modulus operator, which we write  $\%$ . The formula  $i \% n$ , which mathematicians write  $i \bmod n$ , is the remainder when  $i$  is divided by  $n$ . More formally,  $i \% n$  is the natural number less than  $n$  satisfying  $i = q * n + (i \% n)$  for some natural number  $q$ . Let's express this condition mathematically. The *Naturals* module defines *Nat* to be the set of natural numbers, and the assertion that there exists a  $q$  in the set *Nat* satisfying a formula  $F$  is written  $\exists q \in \text{Nat} : F$ . Thus, if  $i$  and  $n$  are elements of *Nat* and  $n > 0$ , then  $i \% n$  is the unique number satisfying

$$(i \% n \in 0 \dots (n - 1)) \wedge (\exists q \in \text{Nat} : i = q * n + (i \% n))$$

We can use  $\%$  to simplify our hour-clock specification a bit. Observing that  $(11 \% 12) + 1$  equals 12 and  $(12 \% 12) + 1$  equals 1, we can define a different next-state action  $HC_{nxt2}$  and a different formula  $HC2$  to be the clock specification

$$HC_{nxt2} \triangleq hr' = (hr \% 12) + 1 \quad HC2 \triangleq HC_{ini} \wedge \square[HC_{nxt2}]_{hr}$$

Actions  $HC_{nxt}$  and  $HC_{nxt2}$  are not equivalent. The step  $[hr = 24] \rightarrow [hr = 25]$  satisfies  $HC_{nxt}$  but not  $HC_{nxt2}$ , while the step  $[hr = 24] \rightarrow [hr = 1]$  satisfies  $HC_{nxt2}$  but not  $HC_{nxt}$ . However, any step starting in a state with  $hr$  in  $1 \dots 12$  satisfies  $HC_{nxt}$  iff it satisfies  $HC_{nxt2}$ . It's therefore not hard to deduce that any behavior starting in a state satisfying  $HC_{ini}$  satisfies  $\square[HC_{nxt}]_{hr}$  iff it satisfies  $\square[HC_{nxt2}]_{hr}$ . Hence, formulas  $HC$  and  $HC2$  are equivalent. In other words,  $HC \equiv HC2$  is a theorem. It doesn't matter which of the two formulas we take to be the specification of an hour clock.

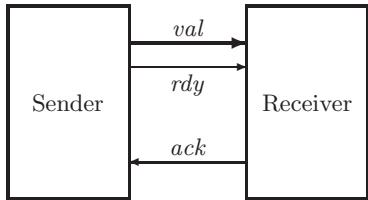
Mathematics provides infinitely many ways of expressing the same thing. The expressions  $6 + 6$ ,  $3 * 4$ , and  $141 - 129$  all have the same meaning; they are just different ways of writing the number 12. We could replace either instance of the number 12 in module *HourClock* by any of these expressions without changing the meaning of any of the module's formulas.

When writing a specification, you will often be faced with a choice of how to express something. When that happens, you should first make sure that the choices yield equivalent specifications. If they do, then you can choose the one that you feel makes the specification easiest to understand. If they don't, then you must decide which one you mean.

# Chapter 3

## An Asynchronous Interface

We now specify an interface for transmitting data between asynchronous devices. A *sender* and a *receiver* are connected as shown here.



Data is sent on *val*, and the *rdy* and *ack* lines are used for synchronization. The sender must wait for an acknowledgment (an *Ack*) for one data item before it can send the next. The interface uses the standard two-phase handshake protocol, described by the following sample behavior:

$$\begin{bmatrix} val = 26 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{\text{Send 37}} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{\text{Ack}} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 1 \end{bmatrix} \xrightarrow{\text{Send 4}} \dots$$
$$\begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 1 \end{bmatrix} \xrightarrow{\text{Ack}} \begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{\text{Send 19}} \begin{bmatrix} val = 19 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{\text{Ack}} \dots$$

(It doesn't matter what value *val* has in the initial state.)

It's easy to see from this sample behavior what the set of all possible behaviors should be—once we decide what the data values are that can be sent. But, before writing the TLA<sup>+</sup> specification that describes these behaviors, let's look at what I've just done.

In writing this behavior, I made the decision that *val* and *rdy* should change in a single step. The values of the variables *val* and *rdy* represent voltages

on some set of wires in the physical device. Voltages on different wires don’t change at precisely the same instant. I decided to ignore this aspect of the physical system and pretend that the values of *val* and *rdy* represented by those voltages change instantaneously. This simplifies the specification, but at the price of ignoring what may be an important detail of the system. In an actual implementation of the protocol, the voltage on the *rdy* line shouldn’t change until the voltages on the *val* lines have stabilized; but you won’t learn that from my specification. Had I wanted the specification to convey this requirement, I would have written a behavior in which the value of *val* and the value of *rdy* change in separate steps.

A specification is an abstraction. It describes some aspects of the system and ignores others. We want the specification to be as simple as possible, so we want to ignore as many details as we can. But, whenever we omit some aspect of the system from the specification, we admit a potential source of error. With my specification, we can verify the correctness of a system that uses this interface, and the system could still fail because the implementer didn’t know that the *val* line should stabilize before the *rdy* line is changed.

The hardest part of writing a specification is choosing the proper abstraction. I can teach you about  $\text{TLA}^+$ , so expressing an abstract view of a system as a  $\text{TLA}^+$  specification becomes a straightforward task. But I don’t know how to teach you about abstraction. A good engineer knows how to abstract the essence of a system and suppress the unimportant details when specifying and designing it. The art of abstraction is learned only through experience.

When writing a specification, you must first choose the abstraction. In a  $\text{TLA}^+$  specification, this means choosing the variables that represent the system’s state and the granularity of the steps that change those variables’ values. Should the *rdy* and *ack* lines be represented as separate variables or as a single variable? Should *val* and *rdy* change in one step, two steps, or an arbitrary number of steps? To help make these choices, I recommend that you start by writing the first few steps of one or two sample behaviors, just as I did at the beginning of this section. Chapter 7 has more to say about these choices.

## 3.1 The First Specification

Let’s specify the asynchronous interface with a module *AsynchInterface*. The specification uses subtraction of natural numbers, so our module EXTENDS the *Naturals* module to incorporate the definition of the subtraction operator “ $-$ ”. We next decide what the possible values of *val* should be—that is, what data values may be sent. We could write a specification that places no restriction on the data values. The specification could allow the sender first to send 37, then to send  $\sqrt{-15}$ , and then to send *Nat* (the entire set of natural numbers). However, any real device can send only a restricted set of values. We could pick

some specific set—for example, 32-bit numbers. However, the protocol is the same regardless of whether it's used to send 32-bit numbers or 128-bit numbers. So, we compromise between the two extremes of allowing anything to be sent and allowing only 32-bit numbers to be sent by assuming only that there is some set *Data* of data values that may be sent. The constant *Data* is a parameter of the specification. It's declared by the statement

CONSTANT *Data*

Our three variables are declared by

VARIABLES *val, rdy, ack*

The keywords VARIABLE and VARIABLES are synonymous, as are CONSTANT and CONSTANTS.

The variable *rdy* can assume any value—for example,  $-1/2$ . That is, there exist states that assign the value  $-1/2$  to *rdy*. When discussing the specification, we usually say that *rdy* can assume only the values 0 and 1. What we really mean is that the value of *rdy* equals 0 or 1 in every state of any behavior satisfying the specification. But a reader of the specification shouldn't have to understand the complete specification to figure this out. We can make the specification easier to understand by telling the reader what values the variables can assume in a behavior that satisfies the specification. We could do this with comments, but I prefer to use a definition like this one:

$$\text{TypeInvariant} \triangleq (\text{val} \in \text{Data}) \wedge (\text{rdy} \in \{0, 1\}) \wedge (\text{ack} \in \{0, 1\})$$

I call the set  $\{0, 1\}$  the *type* of *rdy*, and I call *TypeInvariant* a *type invariant*. Let's define *type* and some other terms more precisely.

- A *state function* is an ordinary expression (one with no prime or  $\square$ ) that can contain variables and constants.
- A *state predicate* is a Boolean-valued state function.
- An *invariant* *Inv* of a specification *Spec* is a state predicate such that  $\text{Spec} \Rightarrow \square \text{Inv}$  is a theorem.
- A variable *v* has *type* *T* in a specification *Spec* iff  $v \in T$  is an invariant of *Spec*.

We can make the definition of *TypeInvariant* easier to read by writing it as follows.

$$\begin{aligned} \text{TypeInvariant} \triangleq & \wedge \text{val} \in \text{Data} \\ & \wedge \text{rdy} \in \{0, 1\} \\ & \wedge \text{ack} \in \{0, 1\} \end{aligned}$$

Each conjunct begins with a  $\wedge$  and must lie completely to the right of that  $\wedge$ . (The conjunct may occupy multiple lines). We use a similar notation for disjunctions. When using this bulleted-list notation, the  $\wedge$ 's or  $\vee$ 's must line up precisely (even in the ASCII input). Because the indentation is significant, we can eliminate parentheses, making this notation especially useful when conjunctions and disjunctions are nested.

The formula *TypeInvariant* will not appear as part of the specification. We do not assume that *TypeInvariant* is an invariant; the specification should imply that it is. In fact, its invariance will be asserted as a theorem.

The initial predicate is straightforward. Initially, *val* can equal any element of *Data*. We can start with *rdy* and *ack* either both 0 or both 1.

$$\begin{aligned} \text{Init} \triangleq & \wedge \text{val} \in \text{Data} \\ & \wedge \text{rdy} \in \{0, 1\} \\ & \wedge \text{ack} = \text{rdy} \end{aligned}$$

Now for the next-state action *Next*. A step of the protocol either sends a value or receives a value. We define separately the two actions *Send* and *Rcv* that describe the sending and receiving of a value. A *Next* step (one satisfying action *Next*) is either a *Send* step or a *Rcv* step, so it is a *Send*  $\vee$  *Rcv* step. Therefore, *Next* is defined to equal *Send*  $\vee$  *Rcv*. Let's now define *Send* and *Rcv*.

We say that action *Send* is *enabled* in a state from which it is possible to take a *Send* step. From the sample behavior above, we see that *Send* is enabled iff *rdy* equals *ack*. Usually, the first question we ask about an action is, when is it enabled? So, the definition of an action usually begins with its enabling condition. The first conjunct in the definition of *Send* is therefore  $\text{rdy} = \text{ack}$ . The next conjuncts tell us what the new values of the variables *val*, *rdy*, and *ack* are. The new value *val'* of *val* can be any element of *Data*—that is, any value satisfying  $\text{val}' \in \text{Data}$ . The value of *rdy* changes from 0 to 1 or from 1 to 0, so *rdy'* equals  $1 - \text{rdy}$  (because  $1 = 1 - 0$  and  $0 = 1 - 1$ ). The value of *ack* is left unchanged.

TLA<sup>+</sup> defines UNCHANGED *v* to mean that the expression *v* has the same value in the old and new states. More precisely, UNCHANGED *v* equals  $v' = v$ , where *v'* is the expression obtained from *v* by priming all its variables. So, we define *Send* by

$$\begin{aligned} \text{Send} \triangleq & \wedge \text{rdy} = \text{ack} \\ & \wedge \text{val}' \in \text{Data} \\ & \wedge \text{rdy}' = 1 - \text{rdy} \\ & \wedge \text{UNCHANGED ack} \end{aligned}$$

(I could have written  $\text{ack}' = \text{ack}$  instead of UNCHANGED *ack*, but I prefer to use the UNCHANGED construct in specifications.)

A *Rcv* step is enabled iff *rdy* is different from *ack*; it complements the value of *ack* and leaves *val* and *rdy* unchanged. Both *val* and *rdy* are left unchanged iff

---

MODULE *AsynchInterface*

---

EXTENDS *Naturals*

CONSTANT *Data*

VARIABLES *val*, *rdy*, *ack*

*TypeInvariant*  $\triangleq$   $\wedge \text{val} \in \text{Data}$   
 $\wedge \text{rdy} \in \{0, 1\}$   
 $\wedge \text{ack} \in \{0, 1\}$

---

*Init*  $\triangleq$   $\wedge \text{val} \in \text{Data}$   
 $\wedge \text{rdy} \in \{0, 1\}$   
 $\wedge \text{ack} = \text{rdy}$

*Send*  $\triangleq$   $\wedge \text{rdy} = \text{ack}$   
 $\wedge \text{val}' \in \text{Data}$   
 $\wedge \text{rdy}' = 1 - \text{rdy}$   
 $\wedge \text{UNCHANGED } \text{ack}$

*Rcv*  $\triangleq$   $\wedge \text{rdy} \neq \text{ack}$   
 $\wedge \text{ack}' = 1 - \text{ack}$   
 $\wedge \text{UNCHANGED } \langle \text{val}, \text{rdy} \rangle$

*Next*  $\triangleq$  *Send*  $\vee$  *Rcv*

*Spec*  $\triangleq$  *Init*  $\wedge$   $\square[\text{Next}]_{\langle \text{val}, \text{rdy}, \text{ack} \rangle}$

---

THEOREM *Spec*  $\Rightarrow$   $\square \text{TypeInvariant}$

---

**Figure 3.1:** Our first specification of an asynchronous interface.

the pair of values *val*, *rdy* is left unchanged. TLA<sup>+</sup> uses angle brackets  $\langle$  and  $\rangle$  to enclose ordered tuples, so *Rcv* asserts that  $\langle \text{val}, \text{rdy} \rangle$  is left unchanged. (Angle brackets are typed in ASCII as `<<` and `>>`.) The definition of *Rcv* is therefore

$$\begin{aligned} \text{Rcv} \triangleq & \wedge \text{rdy} \neq \text{ack} \\ & \wedge \text{ack}' = 1 - \text{ack} \\ & \wedge \text{UNCHANGED } \langle \text{val}, \text{rdy} \rangle \end{aligned}$$

As in our clock example, the complete specification *Spec* should allow stuttering steps—in this case, ones that leave all three variables unchanged. So, *Spec* allows steps that leave  $\langle \text{val}, \text{rdy}, \text{ack} \rangle$  unchanged. Its definition is

$$\text{Spec} \triangleq \text{Init} \wedge \square[\text{Next}]_{\langle \text{val}, \text{rdy}, \text{ack} \rangle}$$

Module *AsynchInterface* also asserts the invariance of *TypeInvariant*. It appears in full in Figure 3.1 on this page.

## 3.2 Another Specification

Module *AsynchInterface* is a fine description of the interface and its handshake protocol. However, it's not well suited for helping to specify systems that use the interface. Let's rewrite the interface specification in a form that makes it more convenient to use as part of a larger specification.

The first problem with the original specification is that it uses three variables to describe a single interface. A system might use several different instances of the interface. To avoid a proliferation of variables, we replace the three variables *val*, *rdy*, *ack* with a single variable *chan* (short for *channel*). A mathematician would do this by letting the value of *chan* be an ordered triple—for example, a state  $[chan = \langle -1/2, 0, 1 \rangle]$  might replace the state with  $val = -1/2$ ,  $rdy = 0$ , and  $ack = 1$ . But programmers have learned that using tuples like this leads to mistakes; it's easy to forget if the *ack* line is represented by the second or third component. TLA<sup>+</sup> therefore provides records in addition to more conventional mathematical notation.

Let's represent the state of the channel as a record with *val*, *rdy*, and *ack* fields. If *r* is such a record, then *r.val* is its *val* field. The type invariant asserts that the value of *chan* is an element of the set of all such records *r* in which *r.val* is an element of the set *Data* and *r.rdy* and *r.ack* are elements of the set  $\{0, 1\}$ . This set of records is written

$$[val : Data, rdy : \{0, 1\}, ack : \{0, 1\}]$$

The fields of a record are not ordered, so it doesn't matter in what order we write them. This same set of records can also be written as

$$[ack : \{0, 1\}, val : Data, rdy : \{0, 1\}]$$

Initially, *chan* can equal any element of this set whose *ack* and *rdy* fields are equal, so the initial predicate is the conjunction of the type invariant and the condition  $chan.ack = chan.rdy$ .

A system that uses the interface may perform an operation that sends some data value *d* and performs some other changes that depend on the value *d*. We'd like to represent such an operation as an action that is the conjunction of two separate actions: one that describes the sending of *d* and the other that describes the other changes. Thus, instead of defining an action *Send* that sends some unspecified data value, we define the action *Send(d)* that sends data value *d*. The next-state action is satisfied by a *Send(d)* step, for some *d* in *Data*, or a *Rcv* step. (The value received by a *Rcv* step equals *chan.val*.) Saying that a step is a *Send(d)* step for some *d* in *Data* means that there exists a *d* in *Data* such that the step satisfies *Send(d)*—in other words, that the step is an  $\exists d \in Data : Send(d)$  step. So we define

$$Next \triangleq (\exists d \in Data : Send(d)) \vee Rcv$$

The  $Send(d)$  action asserts that  $chan'$  equals the record  $r$  such that

$$r.val = d \quad r.rdy = 1 - chan.rdy \quad r.ack = chan.ack$$

This record is written in TLA<sup>+</sup> as

$$[val \mapsto d, \ rdy \mapsto 1 - chan.rdy, \ ack \mapsto chan.ack]$$

(The symbol  $\mapsto$  is typed in ASCII as  $\rightarrow$ .) Since the fields of records are not ordered, this record can just as well be written

$$[ack \mapsto chan.ack, \ val \mapsto d, \ rdy \mapsto 1 - chan.rdy]$$

The enabling condition of  $Send(d)$  is that the  $rdy$  and  $ack$  lines are equal, so we can define

$$\begin{aligned} Send(d) &\triangleq \\ &\wedge chan.rdy = chan.ack \\ &\wedge chan' = [val \mapsto d, \ rdy \mapsto 1 - chan.rdy, \ ack \mapsto chan.ack] \end{aligned}$$

This is a perfectly good definition of  $Send(d)$ . However, I prefer a slightly different one. We can describe the value of  $chan'$  by saying that it is the same as the value of  $chan$  except that its  $val$  field equals  $d$  and its  $rdy$  field equals  $1 - chan.rdy$ . In TLA<sup>+</sup>, we can write this value as

$$[chan \text{ EXCEPT } !.val = d, !.rdy = 1 - chan.rdy]$$

Think of the  $!$  as standing for the new record that the EXCEPT expression forms by modifying  $chan$ . So, the expression can be read as the record  $!$  that is the same as  $chan$  except  $!.val$  equals  $d$  and  $!.rdy$  equals  $1 - chan.rdy$ . In the expression that  $!.rdy$  equals, the symbol  $@$  stands for  $chan.rdy$ , so we can write this EXCEPT expression as

$$[chan \text{ EXCEPT } !.val = d, !.rdy = 1 - @]$$

In general, for any record  $r$ , the expression

$$[r \text{ EXCEPT } !.c_1 = e_1, \dots, !.c_n = e_n]$$

is the record obtained from  $r$  by replacing  $r.c_i$  with  $e_i$ , for each  $i$  in  $1 \dots n$ . An  $@$  in the expression  $e_i$  stands for  $r.c_i$ . Using this notation, we define

$$\begin{aligned} Send(d) &\triangleq \wedge chan.rdy = chan.ack \\ &\wedge chan' = [chan \text{ EXCEPT } !.val = d, !.rdy = 1 - @] \end{aligned}$$

The definition of  $Rcv$  is straightforward. A value can be received when  $chan.rdy$  does not equal  $chan.ack$ , and receiving the value complements  $chan.ack$ :

$$\begin{aligned} Rcv &\triangleq \wedge chan.rdy \neq chan.ack \\ &\wedge chan' = [chan \text{ EXCEPT } !.ack = 1 - @] \end{aligned}$$

The complete specification appears in Figure 3.2 on the next page.

MODULE *Channel*EXTENDS *Naturals*CONSTANT *Data*VARIABLE *chan* $TypeInvariant \triangleq \text{chan} \in [\text{val} : \text{Data}, \text{rdy} : \{0, 1\}, \text{ack} : \{0, 1\}]$  $\text{Init} \triangleq \wedge \text{TypeInvariant}$   
 $\wedge \text{chan.ack} = \text{chan.rdy}$  $\text{Send}(d) \triangleq \wedge \text{chan.rdy} = \text{chan.ack}$   
 $\wedge \text{chan}' = [\text{chan EXCEPT } !.\text{val} = d, !.\text{rdy} = 1 - @]$  $\text{Rcv} \triangleq \wedge \text{chan.rdy} \neq \text{chan.ack}$   
 $\wedge \text{chan}' = [\text{chan EXCEPT } !.\text{ack} = 1 - @]$  $\text{Next} \triangleq (\exists d \in \text{Data} : \text{Send}(d)) \vee \text{Rcv}$  $\text{Spec} \triangleq \text{Init} \wedge \square[\text{Next}]_{\text{chan}}$ THEOREM  $\text{Spec} \Rightarrow \square \text{TypeInvariant}$ **Figure 3.2:** Our second specification of an asynchronous interface.

### 3.3 Types: A Reminder

As defined in Section 3.1, a variable  $v$  has type  $T$  in specification  $\text{Spec}$  iff  $v \in T$  is an invariant of  $\text{Spec}$ . Thus,  $hr$  has type  $1 \dots 12$  in the specification  $HC$  of the hour clock. This assertion does *not* mean that the variable  $hr$  can assume only values in the set  $1 \dots 12$ . A state is an arbitrary assignment of values to variables, so there exist states in which the value of  $hr$  is  $\sqrt{-2}$ . The assertion does mean that, in every behavior satisfying formula  $HC$ , the value of  $hr$  is an element of  $1 \dots 12$ .

If you are used to types in programming languages, it may seem strange that TLA<sup>+</sup> allows a variable to assume any value. Why not restrict our states to ones in which variables have the values of the right type? In other words, why not add a formal type system to TLA<sup>+</sup>? A complete answer would take us too far afield. The question is addressed further in Section 6.2. For now, remember that TLA<sup>+</sup> is an untyped language. Type correctness is just a name for a certain invariance property. Assigning the name *TypeInvariant* to a formula gives it no special status.

## 3.4 Definitions

Let's examine what a definition means. If  $Id$  is a simple identifier like  $Init$  or  $Spec$ , then the definition  $Id \triangleq exp$  defines  $Id$  to be synonymous with the expression  $exp$ . Replacing  $Id$  by  $exp$ , or vice-versa, in any expression does not change the meaning of that expression. This replacement must be done after the expression is parsed, not in the “raw input”. For example, the definition  $x \triangleq a + b$  makes  $x * c$  equal to  $(a + b) * c$ , not to  $a + b * c$ , which equals  $a + (b * c)$ .

The definition of  $Send$  has the form  $Id(p) \triangleq exp$ , where  $Id$  and  $p$  are identifiers. For any expression  $e$ , this defines  $Id(e)$  to be the expression obtained by substituting  $e$  for  $p$  in  $exp$ . For example, the definition of  $Send$  in the *Channel* module defines  $Send(-5)$  to equal

$$\begin{aligned} & \wedge chan.rdy = chan.ack \\ & \wedge chan' = [chan \text{ EXCEPT } !.val = -5, !.rdy = 1 - @] \end{aligned}$$

$Send(e)$  is an expression, for any expression  $e$ . Thus, we can write the formula  $Send(-5) \wedge (chan.ack = 1)$ . The identifier  $Send$  by itself is not an expression, and  $Send \wedge (chan.ack = 1)$  is not a grammatically well-formed string. It's non-syntactic nonsense, like  $a + * b +$ .

We say that  $Send$  is an *operator* that takes a single argument. We define operators that take more than one argument in the obvious way, the general form being

$$(3.1) \quad Id(p_1, \dots, p_n) \triangleq exp$$

where the  $p_i$  are distinct identifiers and  $exp$  is an expression. We can consider defined identifiers like  $Init$  and  $Spec$  to be operators that take no argument, but we generally use *operator* to mean an operator that takes one or more arguments.

I will use the term *symbol* to mean an identifier like  $Send$  or an operator symbol like  $+$ . Every symbol that is used in a specification must either be a built-in operator of  $TLA^+$  (like  $\in$ ) or it must be declared or defined. Every symbol declaration or definition has a *scope* within which the symbol may be used. The scope of a VARIABLE or CONSTANT declaration, and of a definition, is the part of the module that follows it. Thus, we can use  $Init$  in any expression that follows its definition in module *Channel*. The statement EXTENDS *Naturals* extends the scope of symbols like  $+$  defined in the *Naturals* module to the *Channel* module.

The operator definition (3.1) implicitly includes a declaration of the identifiers  $p_1, \dots, p_n$  whose scope is the expression  $exp$ . An expression of the form

$$\exists v \in S : exp$$

has a declaration of  $v$  whose scope is the expression  $exp$ . Thus the identifier  $v$  has a meaning within the expression  $exp$  (but not within the expression  $S$ ).

A symbol cannot be declared or defined if it already has a meaning. The expression

$$(\exists v \in S : \text{exp1}) \wedge (\exists v \in T : \text{exp2})$$

is all right, because neither declaration of  $v$  lies within the scope of the other. Similarly, the two declarations of the symbol  $d$  in the *Channel* module (in the definition of *Send* and in the expression  $\exists d$  in the definition of *Next*) have disjoint scopes. However, the expression

$$(\exists v \in S : (\text{exp1} \wedge \exists v \in T : \text{exp2}))$$

is illegal because the declaration of  $v$  in the second  $\exists v$  lies inside the scope of its declaration in the first  $\exists v$ . Although conventional mathematics and programming languages allow such redeclarations, TLA<sup>+</sup> forbids them because they can lead to confusion and errors.

## 3.5 Comments

Even simple specifications like the ones in modules *AsynchInterface* and *Channel* can be hard to understand from the mathematics alone. That's why I began with an intuitive explanation of the interface. That explanation made it easier for you to understand formula *Spec* in the module, which is the actual specification. Every specification should be accompanied by an informal prose explanation. The explanation may be in an accompanying document, or it may be included as comments in the specification.

Figure 3.3 on the next page shows how the hour clock's specification in module *HourClock* might be explained by comments. In the typeset version, comments are distinguished from the specification itself by the use of a different font. As shown in the figure, TLA<sup>+</sup> provides two ways of writing comments in the ASCII version. A comment may appear anywhere enclosed between  $(*$  and  $*)$ . An end-of-line comment is preceded by  $\backslash*$ . Comments may be nested, so you can comment out a section of a specification by enclosing it between  $(*$  and  $*)$ , even if the section contains comments.

A comment almost always appears on a line by itself or at the end of a line. I put a comment between *HCnxt* and  $\triangleq$  just to show that it can be done.

To save space, I will write few comments in the example specifications. But specifications should have lots of comments. Even if there is an accompanying document describing the system, comments are needed to help the reader understand how the specification formalizes that description.

Comments can help solve a problem posed by the logical structure of a specification. A symbol has to be declared or defined before it can be used. In module *Channel*, the definition of *Spec* has to follow the definition of *Next*, which has to follow the definitions of *Send* and *Rcv*. But it's usually easiest to

## ----- MODULE HourClock -----

This module specifies a digital clock that displays the current hour. It ignores real time, not specifying when the display can change.

EXTENDS *Naturals*

VARIABLE *hr*    Variable *hr* represents the display.

$HCini \triangleq hr \in (1 \dots 12)$     Initially, *hr* can have any value from 1 through 12.

$HCnxt \triangleq$  This is a weird place for a comment.  $\triangleq$

The value of *hr* cycles from 1 through 12.

$hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$

$HC \triangleq HCini \wedge \square[HCnxt]_{hr}$

The complete spec. It permits the clock to stop.

----- THEOREM  $HC \Rightarrow \square HCini$     Type-correctness of the spec. -----

## ----- MODULE HourClock -----

```
(*****)
(* This module specifies a digital clock that displays *)
(* the current hour. It ignores real time, not *)
(* specifying when the display can change. *)
(*****)
```

EXTENDS *Naturals*

VARIABLE *hr*    /\* Variable *hr* represents the display.

$HCini == hr \in (1 \dots 12)$     /\* Initially, *hr* can have any  
/\* value from 1 through 12.

$HCnxt \triangleq$  /\* This is a weird place for a comment. \*/ ==

```
(*****)
(* The value of hr cycles from 1 through 12. *)
(*****)
```

```
(*****)
hr' = IF hr # 12 THEN hr + 1 ELSE 1
```

$HC == HCini \wedge [] [HCnxt]_{hr}$

(\* The complete spec. It permits the clock to stop. \*)

----- THEOREM  $HC \Rightarrow [] HCini$  /\* Type-correctness of the spec. -----

**Figure 3.3:** The hour-clock specification with comments.

understand a top-down description of a system. We would probably first want to read the declarations of *Data* and *chan*, then the definition of *Spec*, then the definitions of *Init* and *Next*, and then the definitions of *Send* and *Rcv*. In other words, we want to read the specification more or less from bottom to top. This is easy enough to do for a module as short as *Channel*; it's inconvenient for longer specifications. We can use comments to guide the reader through a longer specification. For example, we could precede the definition of *Send* in the *Channel* module with the comment

Actions *Send* and *Rcv* below are the disjuncts of the next-state action *Next*.

The module structure also allows us to choose the order in which a specification is read. For example, we can rewrite the hour-clock specification by splitting the *HourClock* module into three separate modules:

*HCVar* A module that declares the variable *hr*.

*HCActions* A module that EXTENDS modules *Naturals* and *HCVar* and defines *HCini* and *HCnxt*.

*HCSpec* A module that EXTENDS module *HCActions*, defines formula *HC*, and asserts the type-correctness theorem.

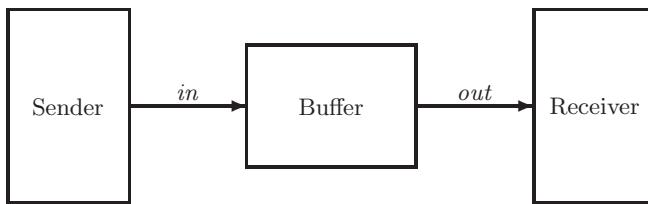
The EXTENDS relation implies a logical ordering of the modules: *HCVar* precedes *HCActions*, which precedes *HCSpec*. But the modules don't have to be read in that order. The reader can be told to read *HCVar* first, then *HCSpec*, and finally *HCActions*. The INSTANCE construct introduced below in Chapter 4 provides another tool for modularizing specifications.

Splitting a tiny specification like *HourClock* in this way would be ludicrous. But the proper splitting of modules can help make a large specification easier to read. When writing a specification, you should decide in what order it should be read. You can then design the module structure to permit reading it in that order, when each individual module is read from beginning to end. Finally, you should ensure that the comments within each module make sense when the different modules are read in the appropriate order.

# Chapter 4

## A FIFO

Our next example is a FIFO buffer, called a FIFO for short—a device with which a sender process transmits a sequence of values to a receiver. The sender and receiver use two channels, *in* and *out*, to communicate with the buffer:



Values are sent over *in* and *out* using the asynchronous protocol specified by the *Channel* module of Figure 3.2 on page 30. The system’s specification will allow behaviors with four kinds of nonstuttering steps: *Send* and *Rcv* steps on both the *in* channel and the *out* channel.

### 4.1 The Inner Specification

The specification of the FIFO first EXTENDS modules *Naturals* and *Sequences*. The *Sequences* module defines operations on finite sequences. We represent a finite sequence as a tuple, so the sequence of three numbers 3, 2, 1 is the triple  $\langle 3, 2, 1 \rangle$ . The *Sequences* module defines the following operators on sequences:

*Seq*(*S*) The set of all sequences of elements of the set *S*. For example,  $\langle 3, 7 \rangle$  is an element of *Seq*(*Nat*).

*Head*(*s*) The first element of sequence *s*. For example, *Head*( $\langle 3, 7 \rangle$ ) equals 3.

*Tail*( $s$ ) The tail of sequence  $s$ , which consists of  $s$  with its head removed.  
 For example, *Tail*( $\langle 3, 7 \rangle$ ) equals  $\langle 7 \rangle$ .

*Append*( $s, e$ ) The sequence obtained by appending element  $e$  to the tail of sequence  $s$ . For example, *Append*( $\langle 3, 7 \rangle, 3$ ) equals  $\langle 3, 7, 3 \rangle$ .

$s \circ t$  The sequence obtained by concatenating the sequences  $s$  and  $t$ . For example,  $\langle 3, 7 \rangle \circ \langle 3 \rangle$  equals  $\langle 3, 7, 3 \rangle$ . (We type  $\circ$  in ASCII as  $\backslash o$ .)

*Len*( $s$ ) The length of sequence  $s$ . For example, *Len*( $\langle 3, 7 \rangle$ ) equals 2.

The FIFO's specification continues by declaring the constant *Message*, which represents the set of all messages that can be sent.<sup>1</sup> It then declares the variables. There are three variables: *in* and *out*, representing the channels, and a third variable *q* that represents the queue of buffered messages. The value of *q* is the sequence of messages that have been sent by the sender but not yet received by the receiver. (Section 4.3 has more to say about this additional variable *q*.)

We want to use the definitions in the *Channel* module to specify operations on the channels *in* and *out*. This requires two instances of that module—one in which the variable *chan* of the *Channel* module is replaced with the variable *in* of our current module, and the other in which *chan* is replaced with *out*. In both instances, the constant *Data* of the *Channel* module is replaced with *Message*. We obtain the first of these instances with the statement

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow in$$

For every symbol  $\sigma$  defined in module *Channel*, this defines *InChan!* $\sigma$  to have the same meaning in the current module as  $\sigma$  had in module *Channel*, except with *Message* substituted for *Data* and *in* substituted for *chan*. For example, this statement defines *InChan!**TypeInvariant* to equal

$$in \in [val : Message, rdy : \{0, 1\}, ack : \{0, 1\}]$$

(The statement does *not* define *InChan!**Data* because *Data* is declared, not defined, in module *Channel*.) We introduce our second instance of the *Channel* module with the analogous statement

$$OutChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow out$$

The initial states of the *in* and *out* channels are specified by *InChan!**Init* and *OutChan!**Init*. Initially, no messages have been sent or received, so *q* should

---

<sup>1</sup>I like to use a singular noun like *Message* rather than a plural like *Messages* for the name of a set. That way, the  $\in$  in the expression  $m \in Message$  can be read *is a*. This is the same convention that most programmers use for naming types.

equal the empty sequence. The empty sequence is the 0-tuple (there's only one, and it's written  $\langle \rangle$ ), so we define the initial predicate to be

$$\begin{aligned} \text{Init} \triangleq & \wedge \text{InChan}!\text{Init} \\ & \wedge \text{OutChan}!\text{Init} \\ & \wedge q = \langle \rangle \end{aligned}$$

We next define the type invariant. The type invariants for *in* and *out* come from the *Channel* module, and the type of *q* is the set of finite sequences of messages. The type invariant for the FIFO specification is therefore

$$\begin{aligned} \text{TypeInvariant} \triangleq & \wedge \text{InChan}!\text{TypeInvariant} \\ & \wedge \text{OutChan}!\text{TypeInvariant} \\ & \wedge q \in \text{Seq}(\text{Message}) \end{aligned}$$

The four kinds of nonstuttering steps allowed by the next-state action are described by four actions:

*SSend*(*msg*) The sender sends message *msg* on the *in* channel.

*BufRcv* The buffer receives the message from the *in* channel and appends it to the tail of *q*.

*BufSend* The buffer removes the message from the head of *q* and sends it on channel *out*.

*RRcv* The receiver receives the message from the *out* channel.

The definitions of these actions, along with the rest of the specification, are in module *InnerFIFO* of Figure 4.1 on the next page. The reason for the adjective *Inner* is explained in Section 4.3 below.

## 4.2 Instantiation Examined

The INSTANCE statement is seldom used except in one idiom for hiding variables, which is described in Section 4.3. So, most readers can skip this section and go directly to page 41.

### 4.2.1 Instantiation Is Substitution

Consider the definition of *Next* in module *Channel* (page 30). We can remove every defined symbol that appears in that definition by using the symbol's definition. For example, we can eliminate the expression *Send*(*d*) by expanding the definition of *Send*. We can repeat this process. For example, the “-” that appears in the expression  $1 - @$  (obtained by expanding the definition of *Send*)

MODULE *InnerFIFO*

EXTENDS *Naturals, Sequences*

CONSTANT *Message*

VARIABLES *in, out, q*

$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow in$

$OutChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow out$

$Init \triangleq \wedge InChan!Init$   
 $\wedge OutChan!Init$   
 $\wedge q = \langle \rangle$

$TypeInvariant \triangleq \wedge InChan!TypeInvariant$   
 $\wedge OutChan!TypeInvariant$   
 $\wedge q \in Seq(Message)$

$SSend(msg) \triangleq \wedge InChan!Send(msg)$  Send *msg* on channel *in*.  
 $\wedge \text{UNCHANGED } \langle out, q \rangle$

$BufRecv \triangleq \wedge InChan!Rcv$  Receive message from channel *in*  
 $\wedge q' = Append(q, in.val)$  and append it to tail of *q*.  
 $\wedge \text{UNCHANGED } out$

$BufSend \triangleq \wedge q \neq \langle \rangle$  Enabled only if *q* is nonempty.  
 $\wedge OutChan!Send(Head(q))$  Send *Head(q)* on channel *out*  
 $\wedge q' = Tail(q)$  and remove it from *q*.  
 $\wedge \text{UNCHANGED } in$

$RRcv \triangleq \wedge OutChan!Rcv$  Receive message from channel *out*.  
 $\wedge \text{UNCHANGED } \langle in, q \rangle$

$Next \triangleq \vee \exists msg \in Message : SSend(msg)$   
 $\vee BufRecv$   
 $\vee BufSend$   
 $\vee RRcv$

$Spec \triangleq Init \wedge \square[Next]_{\langle in, out, q \rangle}$

THEOREM *Spec*  $\Rightarrow$   $\square TypeInvariant$

**Figure 4.1:** The specification of a FIFO, with the internal variable  $q$  visible.

can be eliminated by using the definition of “ $-$ ” from the *Naturals* module. Continuing in this way, we eventually obtain a definition for *Next* in terms of only the built-in operators of  $\text{TLA}^+$  and the parameters *Data* and *chan* of the *Channel* module. We consider this to be the “real” definition of *Next* in module *Channel*. The statement

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow in$$

in module *InnerFIFO* defines *InChan!**Next* to be the formula obtained from this real definition of *Next* by substituting *Message* for *Data* and *in* for *chan*. This defines *InChan!**Next* in terms of only the built-in operators of  $\text{TLA}^+$  and the parameters *Message* and *in* of module *InnerFIFO*.

Let’s now consider an arbitrary INSTANCE statement

$$IM \triangleq \text{INSTANCE } M \text{ WITH } p_1 \leftarrow e_1, \dots, p_n \leftarrow e_n$$

Let  $\Sigma$  be a symbol defined in module *M* and let *d* be its “real” definition. The INSTANCE statement defines *IM!* $\Sigma$  to have as its real definition the expression obtained from *d* by replacing all instances of  $p_i$  by the expression  $e_i$ , for each *i*. The definition of *IM!* $\Sigma$  must contain only the parameters (declared constants and variables) of the current module, not the ones of module *M*. Hence, the  $p_i$  must consist of all the parameters of module *M*. The  $e_i$  must be expressions that are meaningful in the current module.

#### 4.2.2 Parametrized Instantiation

The FIFO specification uses two instances of module *Channel*—one with *in* substituted for *chan* and the other with *out* substituted for *chan*. We could instead use a single parametrized instance by putting the following statement in module *InnerFIFO*:

$$Chan(ch) \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow ch$$

For any symbol  $\Sigma$  defined in module *Channel* and any expression *exp*, this defines *Chan(exp)!* $\Sigma$  to equal formula  $\Sigma$  with *Message* substituted for *Data* and *exp* substituted for *chan*. The *Rcv* action on channel *in* could then be written *Chan(in)!**Rcv*, and the *Send(msg)* action on channel *out* could be written *Chan(out)!**Send(msg)*.

The instantiation above defines *Chan!**Send* to be an operator with two arguments. Writing *Chan(out)!**Send(msg)* instead of *Chan!**Send(out, msg)* is just an idiosyncrasy of the syntax. It is no stranger than the syntax for infix operators, which has us write  $a + b$  instead of  $+(a, b)$ .

Parametrized instantiation is used almost exclusively in the  $\text{TLA}^+$  idiom for variable hiding, described in Section 4.3. You can use that idiom without understanding it, so you probably don’t need to know anything about parametrized instantiation.

### 4.2.3 Implicit Substitutions

The use of *Message* as the name for the set of transmitted values in the FIFO specification is a bit strange, since we had just used the name *Data* for the analogous set in the asynchronous channel specifications. Suppose we had used *Data* in place of *Message* as the constant parameter of module *InnerFIFO*. The first instantiation statement would then have been

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Data, chan \leftarrow in$$

The substitution  $Data \leftarrow Data$  indicates that the constant parameter *Data* of the instantiated module *Channel* is replaced with the expression *Data* of the current module. TLA<sup>+</sup> allows us to drop any substitution of the form  $\Sigma \leftarrow \Sigma$ , for a symbol  $\Sigma$ . So, the statement above can be written as

$$InChan \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow in$$

We know there is an implied  $Data \leftarrow Data$  substitution because an INSTANCE statement must have a substitution for every parameter of the instantiated module. If some parameter *p* has no explicit substitution, then there is an implicit substitution  $p \leftarrow p$ . This means that the INSTANCE statement must lie within the scope of a declaration or definition of the symbol *p*.

It is quite common to instantiate a module with this kind of implicit substitution. Often, every parameter has an implicit substitution, in which case the list of explicit substitutions is empty. The WITH is then omitted.

### 4.2.4 Instantiation Without Renaming

So far, all the instantiations we've used have been with renaming. For example, the first instantiation of module *Channel* renames the defined symbol *Send* as *InChan!Send*. This kind of renaming is necessary if we are using multiple instances of the module, or a single parametrized instance. The two instances *InChan!Init* and *OutChan!Init* of *Init* in module *InnerFIFO* are different formulas, so they need different names.

Sometimes we need only a single instance of a module. For example, suppose we are specifying a system with only a single asynchronous channel. We then need only one instance of *Channel*, so we don't have to rename the instantiated symbols. In that case, we can write something like

$$\text{INSTANCE } Channel \text{ WITH } Data \leftarrow D, chan \leftarrow x$$

This instantiates *Channel* with no renaming, but with substitution. Thus, it defines *Rcv* to be the formula of the same name from the *Channel* module, except with *D* substituted for *Data* and *x* substituted for *chan*. The expressions substituted for an instantiated module's parameters must be defined. So, this INSTANCE statement must be within the scope of the definitions or declarations of *D* and *x*.

## 4.3 Hiding the Queue

Module *InnerFIFO* of Figure 4.1 defines *Spec* to be  $Init \wedge \square[Next] \dots$ , the sort of formula we've become accustomed to as a system specification. However, formula *Spec* describes the value of variable *q*, as well as of the variables *in* and *out*. The picture of the FIFO system I drew on page 35 shows only channels *in* and *out*; it doesn't show anything inside the boxes. A specification of the FIFO should describe only the values sent and received on the channels. The variable *q*, which represents what's going on inside the box labeled *Buffer*, is used to specify what values are sent and received. It is an *internal* variable and, in the final specification, it should be hidden.

In TLA, we hide a variable with the existential quantifier  $\exists$  of temporal logic. The formula  $\exists x : F$  is true of a behavior iff there exists some sequence of values—one in each state of the behavior—that can be assigned to the variable *x* that will make formula *F* true. (The meaning of  $\exists$  is defined more precisely in Section 8.8.)

The obvious way to write a FIFO specification in which *q* is hidden is with the formula  $\exists q : Spec$ . However, we can't put this definition in module *InnerFIFO* because *q* is already declared there, and a formula  $\exists q : \dots$  would redeclare it. Instead, we use a new module with a parametrized instantiation of the *InnerFIFO* module (see Section 4.2.2 on page 39):

---

MODULE *FIFO*

---

```
CONSTANT Message
VARIABLES in, out
Inner(q) \triangleq INSTANCE InnerFIFO
Spec \triangleq $\exists q : \text{Inner}(q) ! Spec$
```

---

Observe that the INSTANCE statement is an abbreviation for

$$\begin{aligned} \text{Inner}(q) &\triangleq \text{INSTANCE InnerFIFO} \\ &\quad \text{WITH } q \leftarrow q, \text{in} \leftarrow \text{in}, \text{out} \leftarrow \text{out}, \text{Message} \leftarrow \text{Message} \end{aligned}$$

The variable parameter *q* of module *InnerFIFO* is instantiated with the parameter *q* of the definition of *Inner*. The other parameters of the *InnerFIFO* module are instantiated with the parameters of module *FIFO*.

If this seems confusing, don't worry about it. Just learn the TLA<sup>+</sup> idiom for hiding variables used here and be content with its intuitive meaning. In fact, for most applications, there's no need to hide variables in the specification. You can just write the inner specification and note in the comments which variables should be regarded as visible and which as internal (hidden).

## 4.4 A Bounded FIFO

We have specified an unbounded FIFO—a buffer that can hold an unbounded number of messages. Any real system has a finite amount of resources, so it can contain only a bounded number of in-transit messages. In many situations, we wish to abstract away the bound on resources and describe a system in terms of unbounded FIFOs. In other situations, we may care about that bound. We then want to strengthen our specification by placing a bound  $N$  on the number of outstanding messages.

A specification of a bounded FIFO differs from our specification of the unbounded FIFO only in that action *BufRcv* should not be enabled unless there are fewer than  $N$  messages in the buffer—that is, unless  $\text{Len}(q)$  is less than  $N$ . It would be easy to write a complete new specification of a bounded FIFO by copying module *InnerFIFO* and just adding the conjunct  $\text{Len}(q) < N$  to the definition of *BufRcv*. But let's use module *InnerFIFO* as it is, rather than copying it.

The next-state action *BNext* for the bounded FIFO is the same as the FIFO's next-state action *Next* except that it allows a *BufRcv* step only if  $\text{Len}(q)$  is less than  $N$ . In other words, *BNext* should allow a step only if (i) it's a *Next* step and (ii) if it's a *BufRcv* step, then  $\text{Len}(q) < N$  is true in the first state. In other words, *BNext* should equal

$$\text{Next} \wedge (\text{BufRcv} \Rightarrow (\text{Len}(q) < N))$$

Module *BoundedFIFO* in Figure 4.2 on the next page contains the specification. It introduces the new constant parameter  $N$ . It also contains the statement

$$\text{ASSUME } (N \in \text{Nat}) \wedge (N > 0)$$

which asserts that, in this module, we are assuming that  $N$  is a positive natural number. Such an assumption has no effect on any definitions made in the module. However, it may be taken as a hypothesis when proving any theorems asserted in the module. In other words, a module asserts that its assumptions imply its theorems. It's a good idea to state this kind of simple assumption about constants.

An *ASSUME* statement should be used only to make assumptions about constants. The formula being assumed should not contain any variables. It might be tempting to assert type declarations as assumptions—for example, to add to module *InnerFIFO* the assumption  $q \in \text{Seq}(\text{Message})$ . However, that would be wrong because it asserts that, in any state,  $q$  is a sequence of messages. As we observed in Section 3.3, a state is a completely arbitrary assignment of values to variables, so there are states in which  $q$  has the value  $\sqrt{-17}$ . Assuming that such a state doesn't exist would lead to a logical contradiction.

You may wonder why module *BoundedFIFO* assumes that  $N$  is a positive natural, but doesn't assume that *Message* is a set. Similarly, why didn't we

MODULE *BoundedFIFO*

```

EXTENDS Naturals, Sequences
VARIABLES in, out
CONSTANT Message, N
ASSUME (N ∈ Nat) ∧ (N > 0)
Inner(q) \triangleq INSTANCE InnerFIFO
BNext(q) \triangleq ∧ Inner(q)! Next
 ∧ Inner(q)! BufRcv \Rightarrow (Len(q) < N)
Spec \triangleq \exists q : Inner(q)! Init \wedge \square [BNext(q)]{in, out, q}
```

**Figure 4.2:** A specification of a FIFO buffer of length *N*.

assume that the constant parameter *Data* in our asynchronous interface specifications is a set? The answer is that, in TLA<sup>+</sup>, every value is a set.<sup>2</sup> A value like the number 3, which we don't think of as a set, is formally a set. We just don't know what its elements are. The formula  $2 \in 3$  is a perfectly reasonable one, but TLA<sup>+</sup> does not specify whether it's true or false. So, we don't have to assume that *Message* is a set because we know that it is one.

Although *Message* is automatically a set, it isn't necessarily a finite set. For example, *Message* could be instantiated with the set *Nat* of natural numbers. If you want to assume that a constant parameter is a finite set, then you need to state this as an assumption. (You can do this with the *IsFiniteSet* operator from the *FiniteSets* module, described in Section 6.1.) However, most specifications make perfect sense for infinite sets of messages or processors, so there is no reason to assume these sets to be finite.

## 4.5 What We're Specifying

I wrote at the beginning of this chapter that we were going to specify a FIFO buffer. Formula *Spec* of the *FIFO* module actually specifies a set of behaviors, each representing a sequence of sending and receiving operations on the channels *in* and *out*. The sending operations on *in* are performed by the sender, and the receiving operations on *out* are performed by the receiver. The sender and receiver are not part of the FIFO buffer; they form its *environment*.

Our specification describes a system consisting of the FIFO buffer and its environment. The behaviors satisfying formula *Spec* of module *FIFO* represent those histories of the universe in which both the system and its environment

<sup>2</sup>TLA<sup>+</sup> is based on the mathematical formalism known as Zermelo-Frankel set theory, also called ZF.

behave correctly. It's often helpful in understanding a specification to indicate explicitly which steps are system steps and which are environment steps. We can do this by defining the next-state action to be

$$Next \triangleq SysNext \vee EnvNext$$

where *SysNext* describes system steps and *EnvNext* describes environment steps. For the FIFO, we have

$$\begin{aligned} SysNext &\triangleq BufRcv \vee BufSend \\ EnvNext &\triangleq (\exists msg \in Message : SSend(msg)) \vee RRcv \end{aligned}$$

While suggestive, this way of defining the next-state action has no formal significance. The specification *Spec* equals  $Init \wedge \square[Next] \dots$ ; changing the way we structure the definition of *Next* doesn't change its meaning. If a behavior fails to satisfy *Spec*, nothing tells us if the system or its environment is to blame.

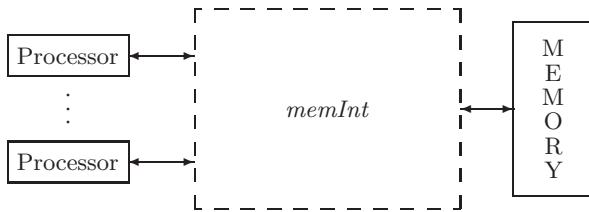
A formula like *Spec*, which describes the correct behavior of both the system and its environment, is called a *closed-system* or *complete-system* specification. An *open-system* specification is one that describes only the correct behavior of the system. A behavior satisfies an open-system specification if it represents a history in which either the system operates correctly, or it failed to operate correctly only because its environment did something wrong. Section 10.7 explains how to write open-system specifications.

Open-system specifications are philosophically more satisfying. However, closed-system specifications are a little easier to write, and the mathematics underlying them is simpler. So, we almost always write closed-system specifications. It's usually quite easy to turn a closed-system specification into an open-system specification. But in practice, there's seldom any reason to do so.

# Chapter 5

## A Caching Memory

A memory system consists of a set of processors connected to a memory by some abstract interface, which we label *memInt*.



In this section we specify what the memory is supposed to do, then we specify a particular implementation of the memory using caches. We begin by specifying the memory interface, which is common to both specifications.

### 5.1 The Memory Interface

The asynchronous interface described in Chapter 3 uses a handshake protocol. Receipt of a data value must be acknowledged before the next data value can be sent. In the memory interface, we abstract away this kind of detail and represent both the sending of a data value and its receipt as a single step. We call it a *Send* step if a processor is sending the value to the memory; it's a *Reply* step if the memory is sending to a processor. Processors do not send values to one another, and the memory sends to only one processor at a time.

We represent the state of the memory interface by the value of the variable *memInt*. A *Send* step changes *memInt* in some way, but we don't want to specify exactly how. The way to leave something unspecified in a specification is to make it a parameter. For example, in the bounded FIFO of Section 4.4, we left the size of the buffer unspecified by making it a parameter *N*. We'd

therefore like to declare a parameter  $Send$  so that  $Send(p, d)$  describes how  $memInt$  is changed by a step that represents processor  $p$  sending data value  $d$  to the memory. However, TLA<sup>+</sup> provides only CONSTANT and VARIABLE parameters, not action parameters.<sup>1</sup> So, we declare  $Send$  to be a constant operator and write  $Send(p, d, memInt, memInt')$  instead of  $Send(p, d)$ .

In TLA<sup>+</sup>, we declare  $Send$  to be a constant operator that takes four arguments by writing

CONSTANT  $Send(\_, \_, \_, \_)$

This means that  $Send(p, d, miOld, miNew)$  is an expression, for any expressions  $p$ ,  $d$ ,  $miOld$ , and  $miNew$ , but it says nothing about what the value of that expression is. We want it to be a Boolean value that is true iff a step in which  $memInt$  equals  $miOld$  in the first state and  $miNew$  in the second state represents the sending by  $p$  of value  $d$  to the memory.<sup>2</sup> We can assert that the value is a Boolean by the assumption

ASSUME  $\forall p, d, miOld, miNew :$   
 $Send(p, d, miOld, miNew) \in \text{BOOLEAN}$

This asserts that the formula

$Send(p, d, miOld, miNew) \in \text{BOOLEAN}$

is true for all values of  $p$ ,  $d$ ,  $miOld$ , and  $miNew$ . The built-in symbol **BOOLEAN** denotes the set  $\{\text{TRUE}, \text{FALSE}\}$ , whose elements are the two Boolean values **TRUE** and **FALSE**.

This ASSUME statement asserts formally that the value of

$Send(p, d, miOld, miNew)$

is a Boolean. But the only way to assert formally what that value signifies would be to say what it actually equals—that is, to define  $Send$  rather than making it a parameter. We don’t want to do that, so we just state informally what the value means. This statement is part of the intrinsically informal description of the relation between our mathematical abstraction and a physical memory system.

To allow the reader to understand the specification, we have to describe informally what  $Send$  means. The ASSUME statement asserting that  $Send(\dots)$  is a Boolean is then superfluous as an explanation. But it’s a good idea to include it anyway.

---

<sup>1</sup>Even if TLA<sup>+</sup> allowed us to declare an action parameter, we would have no way to specify that a  $Send(p, d)$  action constrains only  $memInt$  and not other variables.

<sup>2</sup>We expect  $Send(p, d, miOld, miNew)$  to have this meaning only when  $p$  is a processor and  $d$  a value that  $p$  is allowed to send, but we simplify the specification a bit by requiring it to be a Boolean for all values of  $p$  and  $d$ .

A specification that uses the memory interface can use the operators *Send* and *Reply* to specify how the variable *memInt* changes. The specification must also describe *memInt*'s initial value. We therefore declare a constant parameter *InitMemInt* that is the set of possible initial values of *memInt*.

We also introduce three constant parameters that are needed to describe the interface:

*Proc* The set of processor identifiers. (We usually shorten *processor identifier* to *processor* when referring to an element of *Proc*.)

*Adr* The set of memory addresses.

*Val* The set of possible memory values that can be assigned to an address.

Finally, we define the values that the processors and memory send to one another over the interface. A processor sends a request to the memory. We represent a request as a record with an *op* field that specifies the type of request and additional fields that specify its arguments. Our simple memory allows only read and write requests. A read request has *op* field “Rd” and an *adr* field specifying the address to be read. The set of all read requests is therefore the set

$[op : \{\text{“Rd”}\}, adr : Adr]$

of all records whose *op* field equals “Rd” (is an element of the set  $\{\text{“Rd”}\}$  whose only element is the string “Rd”) and whose *adr* field is an element of *Adr*. A write request must specify the address to be written and the value to write. It is represented by a record with *op* field equal to “Wr”, and with *adr* and *val* fields specifying the address and value. We define *MReq*, the set of all requests, to equal the union of these two sets. (Set operations, including union, are described in Section 1.2 on page 11.)

The memory responds to a read request with the memory value it read. We will also have it respond to a write request, and it seems nice to let the response be different from the response to any read request. We therefore require the memory to respond to a write request by returning a value *NoVal* that is different from any memory value. We could declare *NoVal* to be a constant parameter and add the assumption  $NoVal \notin Val$ . (The symbol  $\notin$  is typed in ASCII as \notin.) But it's best, when possible, to avoid introducing parameters. Instead, we define *NoVal* by

$NoVal \triangleq \text{CHOOSE } v : v \notin Val$

The expression  $\text{CHOOSE } x : F$  equals an arbitrarily chosen value *x* that satisfies the formula *F*. (If no such *x* exists, the expression has a completely arbitrary value.) This statement defines *NoVal* to be some value that is not an element of

| MODULE <i>MemoryInterface</i>                                                                    |                                                                                                            |
|--------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| VARIABLE <i>memInt</i>                                                                           |                                                                                                            |
| CONSTANTS <i>Send</i> ( $\_, \_, \_, \_, \_$ ),                                                  | A <i>Send</i> ( $p, d, memInt, memInt'$ ) step represents processor $p$ sending value $d$ to the memory.   |
| <i>Reply</i> ( $\_, \_, \_, \_, \_$ ),                                                           | A <i>Reply</i> ( $p, d, memInt, memInt'$ ) step represents the memory sending value $d$ to processor $p$ . |
| <i>InitMemInt</i> ,                                                                              | The set of possible initial values of <i>memInt</i> .                                                      |
| <i>Proc</i> ,                                                                                    | The set of processor identifiers.                                                                          |
| <i>Adr</i> ,                                                                                     | The set of memory addresses.                                                                               |
| <i>Val</i>                                                                                       | The set of memory values.                                                                                  |
| ASSUME $\forall p, d, miOld, miNew : \wedge Send(p, d, miOld, miNew) \in \text{BOOLEAN}$         |                                                                                                            |
| $\wedge Reply(p, d, miOld, miNew) \in \text{BOOLEAN}$                                            |                                                                                                            |
| <i>MReq</i> $\triangleq$ $[op : \{“Rd”\}, adr : Adr] \cup [op : \{“Wr”\}, adr : Adr, val : Val]$ | The set of all requests; a read specifies an address, a write specifies an address and a value.            |
| <i>NoVal</i> $\triangleq$ CHOOSE $v : v \notin Val$                                              | An arbitrary value not in <i>Val</i> .                                                                     |

**Figure 5.1:** The specification of a memory interface.

*Val*. We have no idea what the value of *NoVal* is; we just know what it isn’t—namely, that it isn’t an element of *Val*. The CHOOSE operator is discussed in Section 6.6 on page 73.

The complete memory interface specification is module *MemoryInterface* in Figure 5.1 on this page.

## 5.2 Functions

A memory assigns values to addresses. The state of the memory is therefore an assignment of elements of *Val* (memory values) to elements of *Adr* (memory addresses). In a programming language, such an assignment is called an array of type *Val* indexed by *Adr*. In mathematics, it’s called a function from *Adr* to *Val*. Before writing the memory specification, let’s look at the mathematics of functions, and how it is described in TLA<sup>+</sup>.

A function  $f$  has a domain, written DOMAIN  $f$ , and it assigns to each element  $x$  of its domain the value  $f[x]$ . (Mathematicians write this as  $f(x)$ , but TLA<sup>+</sup> uses the array notation of programming languages, with square brackets.) Two functions  $f$  and  $g$  are equal iff they have the same domain and  $f[x] = g[x]$  for all  $x$  in their domain.

The *range* of a function  $f$  is the set of all values of the form  $f[x]$  with  $x$  in DOMAIN  $f$ . For any sets  $S$  and  $T$ , the set of all functions whose domain equals  $S$  and whose range is any subset of  $T$  is written  $[S \rightarrow T]$ .

Ordinary mathematics does not have a convenient notation for writing an expression whose value is a function. TLA<sup>+</sup> defines  $[x \in S \mapsto e]$  to be the function  $f$  with domain  $S$  such that  $f[x] = e$  for every  $x \in S$ .<sup>3</sup> For example,

$$\text{succ} \triangleq [n \in \text{Nat} \mapsto n + 1]$$

defines *succ* to be the successor function on the natural numbers—the function with domain *Nat* such that  $\text{succ}[n] = n + 1$  for all  $n \in \text{Nat}$ .

A record is a function whose domain is a finite set of strings. For example, a record with *val*, *ack*, and *rdy* fields is a function whose domain is the set  $\{\text{"val"}, \text{"ack"}, \text{"rdy"}\}$  consisting of the three strings “val”, “ack”, and “rdy”. The expression  $r.\text{ack}$ , the *ack* field of a record  $r$ , is an abbreviation for  $r[\text{"ack"}]$ . The record

$$[\text{val} \mapsto 42, \text{ack} \mapsto 1, \text{rdy} \mapsto 0]$$

can be written

$$[i \in \{\text{"val"}, \text{"ack"}, \text{"rdy"}\} \mapsto \text{IF } i = \text{"val"} \text{ THEN } 42 \text{ ELSE IF } i = \text{"ack"} \text{ THEN } 1 \text{ ELSE } 0]$$

The EXCEPT construct for records, explained in Section 3.2, is a special case of a general EXCEPT construct for functions, where  $!c$  is an abbreviation for  $!["c"]$ . For any function  $f$ , the expression  $[f \text{ EXCEPT } !c = e]$  is the function  $\hat{f}$  that is the same as  $f$  except with  $\hat{f}[c] = e$ . This function can also be written

$$[x \in \text{DOMAIN } f \mapsto \text{IF } x = c \text{ THEN } e \text{ ELSE } f[x]]$$

assuming that the symbol  $x$  does not occur in any of the expressions  $f$ ,  $c$ , and  $e$ . For example,  $[\text{succ EXCEPT } !42 = 86]$  is the function  $g$  that is the same as *succ* except that  $g[42]$  equals 86 instead of 43.

As in the EXCEPT construct for records, the expression  $e$  in

$$[f \text{ EXCEPT } !c = e]$$

can contain the symbol  $@$ , where it means  $f[c]$ . For example,

$$[\text{succ EXCEPT } !42 = 2 * @] = [\text{succ EXCEPT } !42 = 2 * \text{succ}[42]]$$

In general,

$$[f \text{ EXCEPT } !c_1 = e_1, \dots, !c_n = e_n]$$

---

<sup>3</sup>The  $\in$  in  $[x \in S \mapsto e]$  is just part of the syntax; TLA<sup>+</sup> uses that particular symbol to help you remember what the construct means. Computer scientists write  $\lambda x : S.e$  to represent something similar to  $[x \in S \mapsto e]$ , except that their  $\lambda$  expressions aren't quite the same as the functions of ordinary mathematics that are used in TLA<sup>+</sup>.

is the function  $\hat{f}$  that is the same as  $f$  except with  $\hat{f}[c_i] = e_i$  for each  $i$ . More precisely, this expression equals

$$[\dots [f \text{ EXCEPT } !(c_1) = e_1] \text{ EXCEPT } !(c_2) = e_2] \dots \text{ EXCEPT } !(c_n) = e_n]$$

Functions correspond to the arrays of programming languages. The domain of a function corresponds to the index set of an array. Function  $[f \text{ EXCEPT } !(c) = e]$  corresponds to the array obtained from  $f$  by assigning  $e$  to  $f[c]$ . A function whose range is a set of functions corresponds to an array of arrays. TLA<sup>+</sup> defines  $[f \text{ EXCEPT } !(c)[d] = e]$  to be the function corresponding to the array obtained by assigning  $e$  to  $f[c][d]$ . It can be written as

$$[f \text{ EXCEPT } !(c) = [@ \text{ EXCEPT } !(d) = e]]$$

The generalization to  $[f \text{ EXCEPT } !(c_1) \dots !(c_n) = e]$  for any  $n$  should be obvious. Since a record is a function, this notation can be used for records as well. TLA<sup>+</sup> uniformly maintains the notation that  $\sigma.c$  is an abbreviation for  $\sigma["c"]$ . For example, this implies

$$\begin{aligned} [f \text{ EXCEPT } !(c).d = e] &= [f \text{ EXCEPT } !(c)[\text{"d"}] = e] \\ &= [f \text{ EXCEPT } !(c) = [@ \text{ EXCEPT } !.d = e]] \end{aligned}$$

The TLA<sup>+</sup> definition of records as functions makes it possible to manipulate them in ways that have no counterparts in programming languages. For example, we can define an operator  $R$  such that  $R(r, s)$  is the record obtained from  $r$  by replacing the value of each field  $c$  that is also a field of the record  $s$  with  $s.c$ . In other words, for every field  $c$  of  $r$ , if  $c$  is a field of  $s$  then  $R(r, s).c = s.c$ ; otherwise  $R(r, s).c = r.c$ . The definition is

$$R(r, s) \triangleq [c \in \text{DOMAIN } r \mapsto \text{IF } c \in \text{DOMAIN } s \text{ THEN } s[c] \text{ ELSE } r[c]]$$

So far, we have seen only functions of a single argument, which are the mathematical analog of the one-dimensional arrays of programming languages. Mathematicians also use functions of multiple arguments, which are the analog of multi-dimensional arrays. In TLA<sup>+</sup>, as in ordinary mathematics, a function of multiple arguments is one whose domain is a set of tuples. For example,  $f[5, 3, 1]$  is an abbreviation for  $f[\langle 5, 3, 1 \rangle]$ , the value of the function  $f$  applied to the triple  $\langle 5, 3, 1 \rangle$ .

The function constructs of TLA<sup>+</sup> have extensions for functions of multiple arguments. For example,  $[g \text{ EXCEPT } !(a, b) = e]$  is the function  $\hat{g}$  that is the same as  $g$  except with  $\hat{g}[a, b]$  equal to  $e$ . The expression

$$(5.1) \quad [n \in \text{Nat}, r \in \text{Real} \mapsto n * r]$$

equals the function  $f$  such that  $f[n, r]$  equals  $n * r$ , for all  $n \in \text{Nat}$  and  $r \in \text{Real}$ . Just as  $\forall i \in S : \forall j \in S : P$  can be written as  $\forall i, j \in S : P$ , we can write the function  $[i \in S, j \in S \mapsto e]$  as  $[i, j \in S \mapsto e]$ .

Section 16.1.7 on page 301 describes the general versions of the TLA<sup>+</sup> function constructs for functions with any number of arguments. However, functions of a single argument are all you’re likely to need. You can almost always replace a function of multiple arguments with a function-valued function—for example, writing  $f[a][b]$  instead of  $f[a, b]$ .

## 5.3 A Linearizable Memory

We now specify a very simple memory system in which a processor  $p$  issues a memory request and then waits for a response before issuing the next request. In our specification, the request is executed by accessing (reading or modifying) a variable  $mem$ , which represents the current state of the memory. Because the memory can receive requests from other processors before responding to processor  $p$ , it matters when  $mem$  is accessed. We let the access of  $mem$  occur any time between the request and the response. This specifies what is called a *linearizable* memory. Less restrictive, more practical memory specifications are described in Section 11.2.

In addition to  $mem$ , the specification has the internal variables  $ctl$  and  $buf$ , where  $ctl[p]$  describes the status of processor  $p$ ’s request, and  $buf[p]$  contains either the request or the response. Consider the request  $req$  that equals

$$[op \mapsto \text{``Wr''}, \ adr \mapsto a, \ val \mapsto v]$$

It is a request to write  $v$  to memory address  $a$ , and it generates the response  $NoVal$ . The processing of this request is represented by the following three steps:

$$\begin{array}{c} \left[ \begin{array}{l} ctl[p] = \text{``rdy''} \\ buf[p] = \dots \\ mem[a] = \dots \end{array} \right] \xrightarrow{Req(p)} \left[ \begin{array}{l} ctl[p] = \text{``busy''} \\ buf[p] = req \\ mem[a] = \dots \end{array} \right] \\ \xrightarrow{Do(p)} \left[ \begin{array}{l} ctl[p] = \text{``done''} \\ buf[p] = NoVal \\ mem[a] = v \end{array} \right] \xrightarrow{Rsp(p)} \left[ \begin{array}{l} ctl[p] = \text{``rdy''} \\ buf[p] = NoVal \\ mem[a] = v \end{array} \right] \end{array}$$

A  $Req(p)$  step represents the issuing of a request by processor  $p$ . It is enabled when  $ctl[p] = \text{``rdy''}$ ; it sets  $ctl[p]$  to  $\text{``busy''}$  and sets  $buf[p]$  to the request. A  $Do(p)$  step represents the memory access; it is enabled when  $ctl[p] = \text{``busy''}$  and it sets  $ctl[p]$  to  $\text{``done''}$  and  $buf[p]$  to the response. A  $Rsp(p)$  step represents the memory’s response to  $p$ ; it is enabled when  $ctl[p] = \text{``done''}$  and it sets  $ctl[p]$  to  $\text{``rdy''}$ .

Writing the specification is a straightforward exercise in representing these changes to the variables in TLA<sup>+</sup> notation. The internal specification, with  $mem$ ,  $ctl$ , and  $buf$  visible (free variables), appears in module *InternalMemory* on the following two pages. The memory specification, which hides the three internal variables, is module *Memory* in Figure 5.3 on page 53.

MODULE *InternalMemory*EXTENDS *MemoryInterface*VARIABLES *mem*, *ctl*, *buf* $IInit \triangleq$  The initial predicate

$$\begin{aligned} & \wedge \text{mem} \in [\text{Adr} \rightarrow \text{Val}] \\ & \wedge \text{ctl} = [\text{p} \in \text{Proc} \mapsto \text{"rdy"}] \\ & \wedge \text{buf} = [\text{p} \in \text{Proc} \mapsto \text{NoVal}] \\ & \wedge \text{memInt} \in \text{InitMemInt} \end{aligned}$$

Initially, memory locations have any values in *Val*,  
 each processor is ready to issue requests,  
 each *buf*[*p*] is arbitrarily initialized to *NoVal*,  
 and *memInt* is any element of *InitMemInt*.

 $TypeInvariant \triangleq$  The type-correctness invariant.

$$\begin{aligned} & \wedge \text{mem} \in [\text{Adr} \rightarrow \text{Val}] & \text{mem is a function from Adr to Val.} \\ & \wedge \text{ctl} \in [\text{Proc} \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"done"}\}] & \text{ctl}[p] \text{ equals "rdy", "busy", or "done".} \\ & \wedge \text{buf} \in [\text{Proc} \rightarrow \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}] & \text{buf}[p] \text{ is a request or a response.} \end{aligned}$$

 $Req(p) \triangleq$  Processor *p* issues a request.

$$\wedge \text{ctl}[p] = \text{"rdy"} \quad \text{Enabled iff } p \text{ is ready to issue a request.}$$

$$\wedge \exists \text{req} \in \text{MReq} : \quad \text{For some request req:}$$

$$\begin{aligned} & \wedge \text{Send}(p, \text{req}, \text{memInt}, \text{memInt}') & \text{Send req on the interface.} \\ & \wedge \text{buf}' = [\text{buf} \text{ EXCEPT } ![\text{p}] = \text{req}] & \text{Set buf}[p] to the request. \\ & \wedge \text{ctl}' = [\text{ctl} \text{ EXCEPT } ![\text{p}] = \text{"busy"}] & \text{Set ctl}[p] to "busy". \end{aligned}$$

$$\wedge \text{UNCHANGED } \text{mem}$$

 $Do(p) \triangleq$  Perform *p*'s request to memory.

$$\wedge \text{ctl}[p] = \text{"busy"} \quad \text{Enabled iff } p \text{ 's request is pending.}$$

$$\wedge \text{mem}' = \text{IF } \text{buf}[p].\text{op} = \text{"Wr"}$$

$$\begin{aligned} & \text{THEN } [\text{mem EXCEPT} & \text{Write to memory on a} \\ & ![\text{buf}[p].\text{adr}] = \text{buf}[p].\text{val}] & \text{"Wr" request.} \end{aligned}$$

$$\text{ELSE } \text{mem} \quad \text{Leave mem unchanged on a "Rd" request.}$$

$$\wedge \text{buf}' = [\text{buf} \text{ EXCEPT}$$

$$\begin{aligned} & ![\text{p}] = \text{IF } \text{buf}[p].\text{op} = \text{"Wr"} & \text{Set buf}[p] to the response:} \\ & \text{THEN } \text{NoVal} & \text{NoVal for a write;} \end{aligned}$$

$$\text{ELSE } \text{mem}[\text{buf}[p].\text{adr}]] \quad \text{the memory value for a read.}$$

$$\wedge \text{ctl}' = [\text{ctl} \text{ EXCEPT } ![\text{p}] = \text{"done"}]$$

$$\text{Set ctl}[p] to "done".$$

$$\wedge \text{UNCHANGED } \text{memInt}$$

**Figure 5.2a:** The internal memory specification (beginning).

|                                                                                                                    |                                              |
|--------------------------------------------------------------------------------------------------------------------|----------------------------------------------|
| $Rsp(p) \triangleq$                                                                                                | Return the response to $p$ 's request.       |
| $\wedge \text{ctl}[p] = \text{"done"}$                                                                             | Enabled iff req. is done but resp. not sent. |
| $\wedge \text{Reply}(p, \text{buf}[p], \text{memInt}, \text{memInt}')$                                             | Send the response on the interface.          |
| $\wedge \text{ctl}' = [\text{ctl EXCEPT } !p = \text{"rdy"}]$                                                      | Set $\text{ctl}[p]$ to "rdy".                |
| $\wedge \text{UNCHANGED } \langle \text{mem}, \text{buf} \rangle$                                                  |                                              |
| $INext \triangleq \exists p \in \text{Proc} : \text{Req}(p) \vee \text{Do}(p) \vee Rsp(p)$                         | The next-state action.                       |
| $ISpec \triangleq IInit \wedge \square[INext]_{\langle \text{memInt}, \text{mem}, \text{ctl}, \text{buf} \rangle}$ | The specification.                           |

---

THEOREM  $ISpec \Rightarrow \square \text{TypeInvariant}$

---

**Figure 5.2b:** The internal memory specification (end).

## 5.4 Tuples as Functions

Before writing our caching memory specification, let's take a closer look at tuples. Recall that  $\langle a, b, c \rangle$  is the 3-tuple with components  $a$ ,  $b$ , and  $c$ . In TLA<sup>+</sup>, this 3-tuple is actually the function with domain  $\{1, 2, 3\}$  that maps 1 to  $a$ , 2 to  $b$ , and 3 to  $c$ . Thus,  $\langle a, b, c \rangle[2]$  equals  $b$ .

TLA<sup>+</sup> provides the Cartesian product operator  $\times$  of ordinary mathematics, where  $A \times B \times C$  is the set of all 3-tuples  $\langle a, b, c \rangle$  such that  $a \in A$ ,  $b \in B$ , and  $c \in C$ . Note that  $A \times B \times C$  is different from  $A \times (B \times C)$ , which is the set of pairs  $\langle a, p \rangle$  with  $a$  in  $A$  and  $p$  in the set of pairs  $B \times C$ .

The *Sequences* module defines finite sequences to be tuples. Hence, a sequence of length  $n$  is a function with domain  $1 \dots n$ . In fact,  $s$  is a sequence iff it equals  $[i \in 1 \dots \text{Len}(s) \mapsto s[i]]$ . Below are a few operator definitions from the *Sequences* module. (The meanings of the operators are described in Section 4.1.)

$$\begin{aligned}
 \text{Head}(s) &\triangleq s[1] \\
 \text{Tail}(s) &\triangleq [i \in 1 \dots (\text{Len}(s) - 1) \mapsto s[i + 1]] \\
 s \circ t &\triangleq [i \in 1 \dots (\text{Len}(s) + \text{Len}(t)) \mapsto \\
 &\quad \text{IF } i \leq \text{Len}(s) \text{ THEN } s[i] \text{ ELSE } t[i - \text{Len}(s)]]
 \end{aligned}$$

---

MODULE *Memory*

---

$$\begin{aligned}
 &\text{EXTENDS } \text{MemoryInterface} \\
 \text{Inner}(\text{mem}, \text{ctl}, \text{buf}) &\triangleq \text{INSTANCE InternalMemory} \\
 \text{Spec} &\triangleq \exists \text{mem}, \text{ctl}, \text{buf} : \text{Inner}(\text{mem}, \text{ctl}, \text{buf})!ISpec
 \end{aligned}$$


---

**Figure 5.3:** The memory specification.

## 5.5 Recursive Function Definitions

We need one more tool to write the caching memory specification: recursive function definitions. Recursively defined functions are familiar to programmers. The classic example is the factorial function, which I'll call *fact*. It's usually defined by writing

$$fact[n] = \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1]$$

for all  $n \in Nat$ . The TLA<sup>+</sup> notation for writing functions suggests trying to define *fact* by

$$fact \triangleq [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1]]$$

This definition is illegal because the occurrence of *fact* to the right of the  $\triangleq$  is undefined—*fact* is defined only after its definition.

TLA<sup>+</sup> does allow the apparent circularity of recursive function definitions. We can define the factorial function *fact* by

$$fact[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1]$$

In general, a definition of the form  $f[x \in S] \triangleq e$  can be used to define recursively a function *f* with domain *S*.

The function definition notation has a straightforward generalization to definitions of functions of multiple arguments. For example,

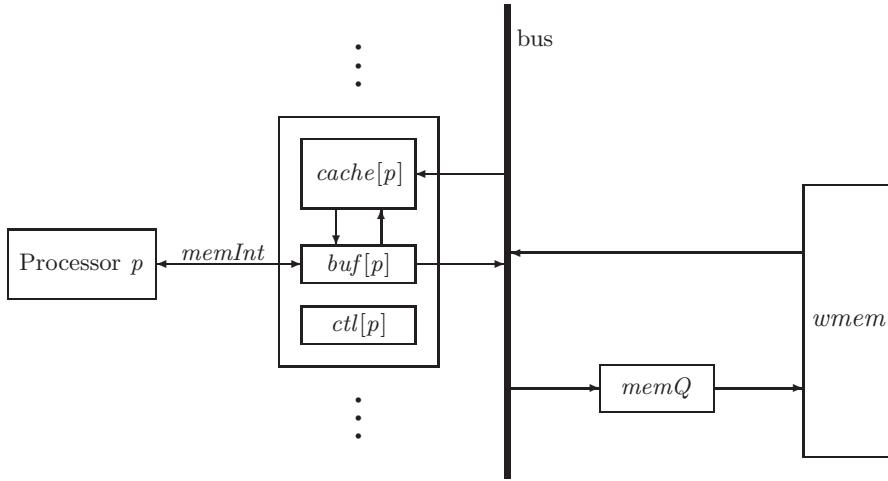
$$\begin{aligned} Acker[m, n \in Nat] &\triangleq \\ &\text{IF } m = 0 \text{ THEN } n + 1 \\ &\text{ELSE IF } n = 0 \text{ THEN } Acker[m - 1, 0] \\ &\qquad\qquad\qquad\text{ELSE } Acker[m - 1, Acker[m, n - 1]] \end{aligned}$$

defines *Acker*[*m*, *n*] for all natural numbers *m* and *n*.

Section 6.3 explains exactly what recursive definitions mean. For now, we will just write recursive definitions without worrying about their meaning.

## 5.6 A Write-Through Cache

We now specify a simple write-through cache that implements the memory specification. The system is described by the picture of Figure 5.4 on the next page. Each processor *p* communicates with a local controller, which maintains three state components: *buf*[*p*], *ctl*[*p*], and *cache*[*p*]. The value of *cache*[*p*] represents the processor's cache; *buf*[*p*] and *ctl*[*p*] play the same role as in the internal memory specification (module *InternalMemory*). (However, as we will see below, *ctl*[*p*] can assume an additional value “waiting”.) These local controllers



**Figure 5.4:** The write-through cache.

communicate with the main memory  $wmem$ ,<sup>4</sup> and with one another, over a bus. Requests from the processors to the main memory are in the queue  $memQ$  of maximum length  $QLen$ .

A write request by processor  $p$  is performed by the action  $DoWr(p)$ . This is a write-through cache, meaning that every write request updates main memory. So, the  $DoWr(p)$  action writes the value into  $cache[p]$  and adds the write request to the tail of  $memQ$ . When the request reaches the head of  $memQ$ , the action  $MemQWr$  stores the value in  $wmem$ . The  $DoWr(p)$  action also updates  $cache[q]$  for every other processor  $q$  that has a copy of the address in its cache.

A read request by processor  $p$  is performed by the action  $DoRd(p)$ , which obtains the value from the cache. If the value is not in the cache, the action  $RdMiss(p)$  adds the request to the tail of  $memQ$  and sets  $ctl[p]$  to “waiting”. When the enqueued request reaches the head of  $memQ$ , the action  $MemQRd$  reads the value and puts it in  $cache[p]$ , enabling the  $DoRd(p)$  action.

We might expect the  $MemQRd$  action to read the value from  $wmem$ . However, this could cause an error if there is a write to that address enqueued in  $memQ$  behind the read request. In that case, reading the value from memory could lead to two processors having different values for the address in their caches: the one that issued the read request, and the one that issued the write request that followed the read in  $memQ$ . So, the  $MemQRd$  action must read the value from the last write to that address in  $memQ$ , if there is such a write; otherwise, it reads the value from  $wmem$ .

<sup>4</sup>We use the name  $wmem$  to distinguish this variable from variable  $mem$  of module *InternalMemory*. We don't have to, since  $mem$  is not a free (visible) variable of the actual memory specification in module *Memory*, but it helps us avoid getting confused.

Eviction of an address from processor  $p$ 's cache is represented by a separate  $Evict(p)$  action. Since all cached values have been written to memory, eviction does nothing but remove the address from the cache. There is no reason to evict an address until the space is needed, so in an implementation, this action would be executed only when a request for an uncached address is received from  $p$  and  $p$ 's cache is full. But that's a performance optimization; it doesn't affect the correctness of the algorithm, so it doesn't appear in the specification. We allow a cached address to be evicted from  $p$ 's cache at any time—except if the address was just put there by a  $MemQRd$  action for a read request whose  $DoRd(p)$  action has not yet been performed. This is the case when  $ctl[p]$  equals “waiting” and  $buf[p].adr$  equals the cached address.

The actions  $Req(p)$  and  $Rsp(p)$ , which represent processor  $p$  issuing a request and the memory issuing a reply to  $p$ , are the same as the corresponding actions of the memory specification, except that they also leave the new variables  $cache$  and  $memQ$  unchanged, and they leave unchanged  $wmem$  instead of  $mem$ .

To specify all these actions, we must decide how the processor caches and the queue of requests to memory are represented by the variables  $memQ$  and  $cache$ . We let  $memQ$  be a sequence of pairs of the form  $\langle p, req \rangle$ , where  $req$  is a request and  $p$  is the processor that issued it. For any memory address  $a$ , we let  $cache[p][a]$  be the value in  $p$ 's cache for address  $a$  (the “copy” of  $a$  in  $p$ 's cache). If  $p$ 's cache does not have a copy of  $a$ , we let  $cache[p][a]$  equal  $NoVal$ .

The specification appears in module *WriteThroughCache* on pages 57–59. I'll now go through this specification, explaining some of the finer points and some notation that we haven't encountered before.

The EXTENDS, declaration statements, and ASSUME are familiar. We can reuse some of the definitions from the *InternalMemory* module, so an INSTANCE statement instantiates a copy of that module with  $wmem$  substituted for  $mem$ . (The other parameters of module *InternalMemory* are instantiated by the parameters of the same name in module *WriteThroughCache*.)

The initial predicate *Init* contains the conjunct  $M!IIinit$ , which asserts that  $ctl$  and  $buf$  have the same initial values as in the internal memory specification, and that  $wmem$  has the same initial value as  $mem$  does in that specification. The write-through cache allows  $ctl[p]$  to have the value “waiting” that it didn't in the internal memory specification, so we can't reuse the internal memory's type invariant  $M!TypeInvariant$ . Formula *TypeInvariant* therefore explicitly describes the types of  $wmem$ ,  $ctl$ , and  $buf$ . The type of  $memQ$  is the set of sequences of  $\langle$ processor, request $\rangle$  pairs.

The module next defines the predicate *Coherence*, which asserts the basic cache coherence property of the write-through cache: for any processors  $p$  and  $q$  and any address  $a$ , if  $p$  and  $q$  both have copies of address  $a$  in their caches, then those copies are equal. Note the trick of writing  $x \notin \{y, z\}$  instead of the equivalent but longer formula  $(x \neq y) \wedge (x \neq z)$ .

MODULE *WriteThroughCache*EXTENDS *Naturals, Sequences, MemoryInterface*VARIABLES *wmem, ctl, buf, cache, memQ*CONSTANT *QLen*ASSUME  $(QLen \in \text{Nat}) \wedge (QLen > 0)$  $M \triangleq \text{INSTANCE InternalMemory WITH } \text{mem} \leftarrow \text{wmem}$ *Init*  $\triangleq$  The initial predicate $\wedge M!Init$  *wmem, buf, and ctl* are initialized as in the internal memory spec. $\wedge \text{cache} =$  All caches are initially empty ( $\text{cache}[p][a] = \text{NoVal}$  for all  $p, a$ ). $[p \in \text{Proc} \mapsto [a \in \text{Adr} \mapsto \text{NoVal}]]$  $\wedge \text{memQ} = \langle \rangle$  The queue *memQ* is initially empty.*TypeInvariant*  $\triangleq$  The type invariant. $\wedge \text{wmem} \in [\text{Adr} \rightarrow \text{Val}]$  $\wedge \text{ctl} \in [\text{Proc} \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"waiting"}, \text{"done"}\}]$  $\wedge \text{buf} \in [\text{Proc} \rightarrow \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}]$  $\wedge \text{cache} \in [\text{Proc} \rightarrow [\text{Adr} \rightarrow \text{Val} \cup \{\text{NoVal}\}]]$  $\wedge \text{memQ} \in \text{Seq}(\text{Proc} \times \text{MReq})$  *memQ* is a sequence of  $\langle \text{proc.}, \text{request} \rangle$  pairs.*Coherence*  $\triangleq$  Asserts that if two processors' caches both have copies $\forall p, q \in \text{Proc}, a \in \text{Adr} : \text{of an address, then those copies have equal values.}$  $(\text{NoVal} \notin \{\text{cache}[p][a], \text{cache}[q][a]\}) \Rightarrow (\text{cache}[p][a] = \text{cache}[q][a])$ *Req(p)*  $\triangleq$  Processor *p* issues a request. $M!Req(p) \wedge \text{UNCHANGED } \langle \text{cache}, \text{memQ} \rangle$ *Rsp(p)*  $\triangleq$  The system issues a response to processor *p*. $M!Rsp(p) \wedge \text{UNCHANGED } \langle \text{cache}, \text{memQ} \rangle$ *RdMiss(p)*  $\triangleq$  Enqueue a request to write value from memory to *p*'s cache. $\wedge (\text{ctl}[p] = \text{"busy"}) \wedge (\text{buf}[p].op = \text{"Rd"})$  Enabled on a read request when $\wedge \text{cache}[p][\text{buf}[p].adr] = \text{NoVal}$  the address is not in *p*'s cache $\wedge \text{Len}(\text{memQ}) < \text{QLen}$  and *memQ* is not full. $\wedge \text{memQ}' = \text{Append}(\text{memQ}, \langle p, \text{buf}[p] \rangle)$  Append  $\langle p, \text{request} \rangle$  to *memQ*. $\wedge \text{ctl}' = [\text{ctl EXCEPT } ![p] = \text{"waiting"}]$  Set *ctl*[*p*] to "waiting". $\wedge \text{UNCHANGED } \langle \text{memInt}, \text{wmem}, \text{buf}, \text{cache} \rangle$ **Figure 5.5a:** The write-through cache specification (beginning).

|                                                                                                  |                                                                                                 |                                |
|--------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|--------------------------------|
| $DoRd(p) \triangleq$                                                                             | Perform a read by $p$ of a value in its cache.                                                  |                                |
| $\wedge \text{ctl}[p] \in \{\text{"busy"}, \text{"waiting"}\}$                                   |                                                                                                 | Enabled if a read              |
| $\wedge \text{buf}[p].op = \text{"Rd"}$                                                          |                                                                                                 | request is pending and         |
| $\wedge \text{cache}[p][\text{buf}[p].adr] \neq \text{NoVal}$                                    |                                                                                                 | address is in cache.           |
| $\wedge \text{buf}' = [\text{buf} \text{ EXCEPT } !(p) = \text{cache}[p][\text{buf}[p].adr]]$    |                                                                                                 | Get result from cache.         |
| $\wedge \text{ctl}' = [\text{ctl} \text{ EXCEPT } !(p) = \text{"done"}]$                         |                                                                                                 | Set $\text{ctl}[p]$ to "done". |
| $\wedge \text{UNCHANGED } \langle \text{memInt}, \text{wmem}, \text{cache}, \text{memQ} \rangle$ |                                                                                                 |                                |
| $DoWr(p) \triangleq$                                                                             | Write to $p$ 's cache, update other caches, and enqueue memory update.                          |                                |
| LET $r \triangleq \text{buf}[p]$                                                                 | Processor $p$ 's request.                                                                       |                                |
| IN $\wedge (\text{ctl}[p] = \text{"busy"}) \wedge (r.op = \text{"Wr"})$                          | Enabled if write request pending                                                                |                                |
| $\wedge \text{Len}(\text{memQ}) < QLen$                                                          | and $\text{memQ}$ is not full.                                                                  |                                |
| $\wedge \text{cache}' =$                                                                         | Update $p$ 's cache and any other cache that has a copy.                                        |                                |
|                                                                                                  | $[q \in \text{Proc} \mapsto \text{IF } (p = q) \vee (\text{cache}[q][r.adr] \neq \text{NoVal})$ |                                |
|                                                                                                  | $\text{THEN } [\text{cache}[q] \text{ EXCEPT } !(r.adr) = r.val]$                               |                                |
|                                                                                                  | $\text{ELSE } \text{cache}[q]]$                                                                 |                                |
| $\wedge \text{memQ}' = \text{Append}(\text{memQ}, \langle p, r \rangle)$                         | Enqueue write at tail of $\text{memQ}$ .                                                        |                                |
| $\wedge \text{buf}' = [\text{buf} \text{ EXCEPT } !(p) = \text{NoVal}]$                          | Generate response.                                                                              |                                |
| $\wedge \text{ctl}' = [\text{ctl} \text{ EXCEPT } !(p) = \text{"done"}]$                         | Set $\text{ctl}$ to indicate request is done.                                                   |                                |
| $\wedge \text{UNCHANGED } \langle \text{memInt}, \text{wmem} \rangle$                            |                                                                                                 |                                |
| $\text{wmem} \triangleq$                                                                         | The value $\text{wmem}$ will have after all the writes in $\text{memQ}$ are performed.          |                                |
| LET $f[i \in 0 \dots \text{Len}(\text{memQ})] \triangleq$                                        | The value $\text{wmem}$ will have after the first                                               |                                |
| IF $i = 0$ THEN $\text{wmem}$                                                                    | $i$ writes in $\text{memQ}$ are performed.                                                      |                                |
| ELSE IF $\text{memQ}[i][2].op = \text{"Rd"}$                                                     |                                                                                                 |                                |
| THEN $f[i - 1]$                                                                                  |                                                                                                 |                                |
| ELSE $[f[i - 1] \text{ EXCEPT } ![\text{memQ}[i][2].adr] =$                                      |                                                                                                 |                                |
| $\text{memQ}[i][2].val]$                                                                         |                                                                                                 |                                |
| IN $f[\text{Len}(\text{memQ})]$                                                                  |                                                                                                 |                                |
| $\text{MemQWr} \triangleq$                                                                       | Perform write at head of $\text{memQ}$ to memory.                                               |                                |
| LET $r \triangleq \text{Head}(\text{memQ})[2]$                                                   | The request at the head of $\text{memQ}$ .                                                      |                                |
| IN $\wedge (\text{memQ} \neq \langle \rangle) \wedge (r.op = \text{"Wr"})$                       | Enabled if $\text{Head}(\text{memQ})$ a write.                                                  |                                |
| $\wedge \text{wmem}' =$                                                                          | Perform the write to memory.                                                                    |                                |
| $[\text{wmem} \text{ EXCEPT } !(r.adr) = r.val]$                                                 |                                                                                                 |                                |
| $\wedge \text{memQ}' = \text{Tail}(\text{memQ})$                                                 | Remove the write from $\text{memQ}$ .                                                           |                                |
| $\wedge \text{UNCHANGED } \langle \text{memInt}, \text{buf}, \text{ctl}, \text{cache} \rangle$   |                                                                                                 |                                |

**Figure 5.5b:** The write-through cache specification (middle).

|                                                                                                   |                                                             |
|---------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| $MemQRd \triangleq$                                                                               | Perform an enqueued read to memory.                         |
| LET $p \triangleq Head(memQ)[1]$                                                                  | The requesting processor.                                   |
| $r \triangleq Head(memQ)[2]$                                                                      | The request at the head of $memQ$ .                         |
| IN $\wedge (memQ \neq \langle \rangle) \wedge (r.op = "Rd")$                                      | Enabled if $Head(memQ)$ is a read.                          |
| $\wedge memQ' = Tail(memQ)$                                                                       | Remove the head of $memQ$ .                                 |
| $\wedge cache' =$                                                                                 | Put value from memory or $memQ$ in $p$ 's cache.            |
|                                                                                                   | [ $cache$ EXCEPT $![p][r.adr] = vmem[r.adr]$ ]              |
| $\wedge$ UNCHANGED $\langle memInt, wmem, buf, ctl \rangle$                                       |                                                             |
| $Evict(p, a) \triangleq$                                                                          | Remove address $a$ from $p$ 's cache.                       |
| $\wedge (ctl[p] = "waiting") \Rightarrow (buf[p].adr \neq a)$                                     | Can't evict $a$ if it was just read into cache from memory. |
| $\wedge cache' = [cache$ EXCEPT $![p][a] = NoVal$ ]                                               |                                                             |
| $\wedge$ UNCHANGED $\langle memInt, wmem, buf, ctl, memQ \rangle$                                 |                                                             |
| $Next \triangleq \vee \exists p \in Proc : \vee Req(p) \vee Rsp(p)$                               |                                                             |
|                                                                                                   | $\vee RdMiss(p) \vee DoRd(p) \vee DoWr(p)$                  |
|                                                                                                   | $\vee \exists a \in Addr : Evict(p, a)$                     |
| $\vee MemQWr \vee MemQRd$                                                                         |                                                             |
| $Spec \triangleq Init \wedge \square[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle}$ |                                                             |
| THEOREM $Spec \Rightarrow \square(TypeInvariant \wedge Coherence)$                                |                                                             |
| $LM \triangleq$ INSTANCE $Memory$                                                                 | The memory spec. with internal variables hidden.            |
| THEOREM $Spec \Rightarrow LM!Spec$                                                                | Formula $Spec$ implements the memory spec.                  |

**Figure 5.5c:** The write-through cache specification (end).

The actions  $Req(p)$  and  $Rsp(p)$ , which represent a processor sending a request and receiving a reply, are essentially the same as the corresponding actions in module *InternalMemory*. However, they must also specify that the variables  $cache$  and  $memQ$ , not present in module *InternalMemory*, are left unchanged.

In the definition of  $RdMiss$ , the expression  $Append(memQ, \langle p, buf[p] \rangle)$  is the sequence obtained by appending the element  $\langle p, buf[p] \rangle$  to the end of  $memQ$ .

The  $DoRd(p)$  action represents the performing of the read from  $p$ 's cache. If  $ctl[p] = "busy"$ , then the address was originally in the cache. If  $ctl[p] = "waiting"$ , then the address was just read into the cache from memory.

The  $DoWr(p)$  action writes the value to  $p$ 's cache and updates the value in any other caches that have copies. It also enqueues a write request in  $memQ$ . In an implementation, the request is put on the bus, which transmits it to the other caches and to the  $memQ$  queue. In our high-level view of the system, we represent all this as a single step.

The definition of *DoWr* introduces the TLA<sup>+</sup> LET/IN construct. The LET clause consists of a sequence of definitions whose scope extends until the end of the IN clause. In the definition of *DoWr*, the LET clause defines  $r$  to equal  $buf[p]$  within the IN clause. Observe that the definition of  $r$  contains the parameter  $p$  of the definition of *DoWr*. Hence, we could not move the definition of  $r$  outside the definition of *DoWr*.

A definition in a LET is just like an ordinary definition in a module; in particular, it can have parameters. These local definitions can be used to shorten an expression by replacing common subexpressions with an operator. In the definition of *DoWr*, I replaced five instances of  $buf[p]$  by the single symbol  $r$ . This was a silly thing to do, because it makes almost no difference in the length of the definition and it requires the reader to remember the definition of the new symbol  $r$ . But using a LET to eliminate common subexpressions can often greatly shorten and simplify an expression.

A LET can also be used to make an expression easier to read, even if the operators it defines appear only once in the IN expression. We write a specification with a sequence of definitions, instead of just defining a single monolithic formula, because a formula is easier to understand when presented in smaller chunks. The LET construct allows the process of splitting a formula into smaller parts to be done hierarchically. A LET can appear as a subexpression of an IN expression. Nested LETs are common in large, complicated specifications.

Next comes the definition of the state function *vmem*, which is used in defining action *MemQRd* below. It equals the value that the main memory *wmem* will have after all the write operations currently in *memQ* have been performed. Recall that the value read by *MemQRd* must be the most recent one written to that address—a value that may still be in *memQ*. That value is the one in *vmem*. The function *vmem* is defined in terms of the recursively defined function  $f$ , where  $f[i]$  is the value *wmem* will have after the first  $i$  operations in *memQ* have been performed. Note that *memQ[i][2]* is the second component (the request) of *memQ[i]*, the  $i^{\text{th}}$  element in the sequence *memQ*.

The next two actions, *MemQWr* and *MemQRd*, represent the processing of the request at the head of the *memQ* queue—*MemQWr* for a write request, and *MemQRd* for a read request. These actions also use a LET to make local definitions. Here, the definitions of  $p$  and  $r$  could be moved before the definition of *MemQWr*. In fact, we could save space by replacing the two local definitions of  $r$  with one global (within the module) definition. However, making the definition of  $r$  global in this way would be somewhat distracting, since  $r$  is used only in the definitions of *MemQWr* and *MemQRd*. It might be better instead to combine these two actions into one. Whether you put a definition into a LET or make it more global should depend on what makes the specification easier to read.

The *Evict(p, a)* action represents the operation of removing address  $a$  from processor  $p$ 's cache. As explained above, we allow an address to be evicted at any time—unless the address was just written to satisfy a pending read request,

which is the case iff  $ctl[p] = \text{“waiting”}$  and  $buf[p].adr = a$ . Note the use of the “double subscript” in the EXCEPT expression of the action’s second conjunct. This conjunct “assigns  $NoVal$  to  $cache[p][a]$ ”. If address  $a$  is not in  $p$ ’s cache, then  $cache[p][a]$  already equals  $NoVal$  and an  $Evict(p, a)$  step is a stuttering step.

The definitions of the next-state action  $Next$  and of the complete specification  $Spec$  are straightforward. The module closes with two theorems that are discussed next.

## 5.7 Invariance

Module *WriteThroughCache* contains the theorem

THEOREM  $Spec \Rightarrow \square(TypeInvariant \wedge Coherence)$

which asserts that  $TypeInvariant \wedge Coherence$  is an invariant of  $Spec$ . A state predicate  $P \wedge Q$  is always true iff both  $P$  and  $Q$  are always true, so  $\square(P \wedge Q)$  is equivalent to  $\square P \wedge \square Q$ . This implies that the theorem above is equivalent to the two theorems

THEOREM  $Spec \Rightarrow \square TypeInvariant$

THEOREM  $Spec \Rightarrow \square Coherence$

The first theorem is the usual type-invariance assertion. The second, which asserts that  $Coherence$  is an invariant of  $Spec$ , expresses an important property of the algorithm.

Although  $TypeInvariant$  and  $Coherence$  are both invariants of the temporal formula  $Spec$ , they differ in a fundamental way. If  $s$  is any state satisfying  $TypeInvariant$ , then any state  $t$  such that  $s \rightarrow t$  is a  $Next$  step also satisfies  $TypeInvariant$ . This property is expressed by

THEOREM  $TypeInvariant \wedge Next \Rightarrow TypeInvariant'$

(Recall that  $TypeInvariant'$  is the formula obtained by priming all the variables in formula  $TypeInvariant$ .) In general, when  $P \wedge N \Rightarrow P'$  holds, we say that predicate  $P$  is an invariant of action  $N$ . Predicate  $TypeInvariant$  is an invariant of  $Spec$  because it is an invariant of  $Next$  and it is implied by the initial predicate  $Init$ .

Predicate  $Coherence$  is not an invariant of the next-state action  $Next$ . For example, suppose  $s$  is a state in which

- $cache[p1][a] = 1$
- $cache[q][b] = NoVal$ , for all  $\langle q, b \rangle$  different from  $\langle p1, a \rangle$
- $wmem[a] = 2$
- $memQ$  contains the single element  $\langle p2, [op \mapsto \text{“Rd”}, adr \mapsto a] \rangle$

An invariant of a specification  $S$  that is also an invariant of its next-state action is sometimes called an *inductive* invariant of  $S$ .

for two different processors  $p1$  and  $p2$  and some address  $a$ . Such a state  $s$  (an assignment of values to variables) exists, assuming that there are at least two processors and at least one address. Then *Coherence* is true in state  $s$ . Let  $t$  be the state obtained from  $s$  by taking a *MemQRd* step. In state  $t$ , we have  $cache[p2][a] = 2$  and  $cache[p1][a] = 1$ , so *Coherence* is false. Hence *Coherence* is not an invariant of the next-state action.

*Coherence* is an invariant of formula *Spec* because states like  $s$  cannot occur in a behavior satisfying *Spec*. Proving its invariance is not so easy. We must find a predicate *Inv* that is an invariant of *Next* such that *Inv* implies *Coherence* and is implied by the initial predicate *Init*.

Important properties of a specification can often be expressed as invariants. Proving that a state predicate  $P$  is an invariant of a specification means proving a formula of the form

$$Init \wedge \square[Next]_v \Rightarrow \square P$$

This is done by finding an appropriate state predicate *Inv* and proving

$$Init \Rightarrow Inv, \quad Inv \wedge [Next]_v \Rightarrow Inv', \quad Inv \Rightarrow P$$

Since our subject is specification, not proof, I won't discuss how to find *Inv*.

## 5.8 Proving Implementation

Module *WriteThroughCache* ends with the theorem

$$\text{THEOREM } Spec \Rightarrow LM!Spec$$

where  $LM!Spec$  is formula *Spec* of module *Memory*. This theorem asserts that every behavior satisfying specification *Spec* of the write-through cache also satisfies  $LM!Spec$ , the specification of a linearizable memory. In other words, it asserts that the write-through cache implements a linearizable memory. In TLA, implementation is implication. A system described by a formula *Sys* implements a specification *Spec* iff *Sys* implies *Spec*—that is, iff  $Sys \Rightarrow Spec$  is a theorem. TLA makes no distinction between system descriptions and specifications; they are both just formulas.

By definition of formula *Spec* of the *Memory* module (page 53), we can restate the theorem as

$$\text{THEOREM } Spec \Rightarrow \exists mem, ctl, buf : LM!Inner(mem, ctl, buf)!ISpec$$

where  $LM!Inner(mem, ctl, buf)!ISpec$  is formula *ISpec* of the *InternalMemory* module. The rules of logic tell us that to prove such a theorem, we must find “witnesses” for the quantified variables *mem*, *ctl*, and *buf*. These witnesses are

state functions (ordinary expressions with no primes), which I'll call  $omem$ ,  $octl$ , and  $obuf$ , that satisfy

$$(5.2) \quad Spec \Rightarrow LM!Inner(omem, octl, obuf)!ISpec$$

Formula  $LM!Inner(omem, octl, obuf)!ISpec$  is formula  $ISpec$  with the substitutions

$$mem \leftarrow omem, \quad ctl \leftarrow octl, \quad buf \leftarrow obuf$$

The tuple  $\langle omem, octl, obuf \rangle$  of witness functions is called a *refinement mapping*, and we describe (5.2) as the assertion that  $Spec$  implements formula  $ISpec$  under this refinement mapping. Intuitively, this means  $Spec$  implies that the value of the tuple  $\langle memInt, omem, octl, obuf \rangle$  of state functions changes the way  $ISpec$  asserts that the tuple  $\langle memInt, mem, ctl, buf \rangle$  of variables should change.

I will now briefly describe how we prove (5.2); for details, see the technical papers about TLA, available through the TLA Web page. Let me first introduce a bit of non-TLA<sup>+</sup> notation. For any formula  $F$  of module *InternalMemory*, let  $\overline{F}$  equal  $LM!Inner(omem, octl, obuf)!F$ , which is formula  $F$  with  $omem$ ,  $octl$ , and  $obuf$  substituted for  $mem$ ,  $ctl$ , and  $buf$ . In particular,  $\overline{mem}$ ,  $\overline{ctl}$ , and  $\overline{buf}$  equal  $omem$ ,  $octl$ , and  $obuf$ , respectively.

With this notation, we can write (5.2) as  $Spec \Rightarrow \overline{ISpec}$ . Replacing  $Spec$  and  $ISpec$  by their definitions, this formula becomes

$$(5.3) \quad Init \wedge \square[Next]_{\langle memInt, wmem, buf, ctl, cache, memQ \rangle}$$

$$\Rightarrow \overline{Init} \wedge \square[\overline{Next}]_{\langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle}$$

$\overline{memInt}$  equals  $memInt$ , since  $memInt$  is a variable distinct from  $mem$ ,  $ctl$ , and  $buf$ .

Formula (5.3) is then proved by finding an invariant  $Inv$  of  $Spec$  such that

$$\begin{aligned} & \wedge Init \Rightarrow \overline{Init} \\ & \wedge Inv \wedge Next \Rightarrow \vee \overline{Next} \\ & \quad \vee \text{UNCHANGED } \langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle \end{aligned}$$

The second conjunct is called *step simulation*. It asserts that a *Next* step starting in a state satisfying the invariant  $Inv$  is either an  $\overline{INext}$  step—a step that changes the 4-tuple  $\langle memInt, omem, octl, obuf \rangle$  the way an  $INext$  step changes  $\langle memInt, mem, ctl, buf \rangle$ —or else it leaves that 4-tuple unchanged. For our memory specifications, the state functions  $omem$ ,  $octl$ , and  $obuf$  are defined by

$$\begin{aligned} omem & \triangleq vmem \\ octl & \triangleq [p \in Proc \mapsto \text{IF } ctl[p] = \text{"waiting"} \text{ THEN } \text{"busy"} \text{ ELSE } ctl[p]] \\ obuf & \triangleq buf \end{aligned}$$

The mathematics of an implementation proof is simple, so the proof is straightforward—in theory. For specifications of real systems, such proofs can be quite difficult. Going from theory to practice requires turning the mathematics

of proofs into an engineering discipline. This is a subject that deserves a book to itself, and I won't try to discuss it here.

You will probably never prove that one specification implements another. However, you should understand refinement mappings and step simulation. You will then be able to use TLC to check that one specification implements another; Chapter 14 explains how.

# Chapter 6

## Some More Math

The mathematics we use to write specifications is built on a small, simple collection of concepts. You've already seen most of what's needed to describe almost any kind of mathematics. All you lack is a handful of operators on sets that are described below in Section 6.1. After learning about them, you will be able to define all the data structures and operations that occur in specifications.

While our mathematics is simple, its foundations are nonobvious—for example, the meanings of recursive function definitions and the CHOOSE operator are subtle. This section discusses some of those foundations. Understanding them will help you use TLA<sup>+</sup> more effectively.

### 6.1 Sets

The simple operations on sets described in Section 1.2 are all you need to write most system specifications. However, you may occasionally have to use more sophisticated operators—especially if you need to define data structures beyond tuples, records, and simple functions.

Two powerful operators of set theory are the unary operators UNION and SUBSET, defined as follows:

UNION  $S$  The union of the elements of  $S$ . In other words, a value  $e$  is an element of UNION  $S$  iff it is an element of an element of  $S$ . For example:

$$\text{UNION } \{\{1, 2\}, \{2, 3\}, \{3, 4\}\} = \{1, 2, 3, 4\}$$

Mathematicians  
write UNION  $S$  as  
 $\bigcup S$ .

SUBSET  $S$  The set of all subsets of  $S$ . In other words,  $T \in \text{SUBSET } S$  iff  $T \subseteq S$ . For example:

$$\text{SUBSET } \{1, 2\} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$$

Mathematicians  
call SUBSET  $S$  the  
*power set* of  $S$   
and write it  $\mathcal{P}(S)$   
or  $2^S$ .

Mathematicians often describe a set as “the set of all … such that …”. TLA<sup>+</sup> has two constructs that formalize such a description:

- $\{x \in S : p\}$  The subset of  $S$  consisting of all elements  $x$  satisfying property  $p$ . For example, the set of odd natural numbers can be written  $\{n \in \text{Nat} : n \% 2 = 1\}$ . The identifier  $x$  is bound in  $p$ ; it may not occur in  $S$ .
- $\{e : x \in S\}$  The set of elements of the form  $e$ , for all  $x$  in the set  $S$ . For example,  $\{2 * n + 1 : n \in \text{Nat}\}$  is the set of all odd natural numbers. The identifier  $x$  is bound in  $e$ ; it may not occur in  $S$ .

The modulus operator `%` is described in Section 2.5 on page 21.

The construct  $\{e : x \in S\}$  has the same generalizations as  $\exists x \in S : F$ . For example,  $\{e : x \in S, y \in T\}$  is the set of all elements of the form  $e$ , for  $x$  in  $S$  and  $y$  in  $T$ . In the construct  $\{x \in S : P\}$ , we can let  $x$  be a tuple. For example,  $\{\langle y, z \rangle \in S : P\}$  is the set of all pairs  $\langle y, z \rangle$  in the set  $S$  that satisfy  $P$ . The grammar of TLA<sup>+</sup> in Chapter 15 specifies precisely what set expressions you can write.

All the set operators we’ve seen so far are built-in operators of TLA<sup>+</sup>. There is also a standard module *FiniteSets* that defines two operators:

*Cardinality*( $S$ ) The number of elements in set  $S$ , if  $S$  is a finite set.

*IsFiniteSet*( $S$ ) True iff  $S$  is a finite set.

The *FiniteSets* module appears on 341. The definition of *Cardinality* is discussed below on page 70.

Careless reasoning about sets can lead to problems. The classic example of this is Russell’s paradox:

Let  $\mathcal{R}$  be the set of all sets  $S$  such that  $S \notin S$ . The definition of  $\mathcal{R}$  implies that  $\mathcal{R} \in \mathcal{R}$  is true iff  $\mathcal{R} \notin \mathcal{R}$  is true.

The formula  $\mathcal{R} \notin \mathcal{R}$  is the negation of  $\mathcal{R} \in \mathcal{R}$ , and a formula and its negation can neither both be true nor both be false. The source of the paradox is that  $\mathcal{R}$  isn’t a set. There’s no way to write it in TLA<sup>+</sup>. Intuitively,  $\mathcal{R}$  is too big to be a set. A collection  $\mathcal{C}$  is too big to be a set if it is as big as the collection of all sets—meaning that we can assign to every set a different element of  $\mathcal{C}$ . That is,  $\mathcal{C}$  is too big to be a set if we can define an operator *SMap* such that

- $\text{SMap}(S)$  is in  $\mathcal{C}$ , for any set  $S$ .
- If  $S$  and  $T$  are two different sets, then  $\text{SMap}(S) \neq \text{SMap}(T)$ .

For example, the collection of all sequences of length 2 is too big to be a set; we can define the operator *SMap* by

$$\text{SMap}(S) \triangleq \langle 1, S \rangle$$

This operator assigns to every set  $S$  a different sequence of length 2.

## 6.2 Silly Expressions

Most modern programming languages introduce some form of type checking to prevent you from writing silly expressions like  $3/\text{"abc"}$ . TLA<sup>+</sup> is based on the usual formalization of mathematics by mathematicians, which doesn't have types. In an untyped formalism, every syntactically well-formed expression has a meaning—even a silly expression like  $3/\text{"abc"}$ . Mathematically, the expression  $3/\text{"abc"}$  is no sillier than the expression  $3/0$ , and mathematicians implicitly write that silly expression all the time. For example, consider the true formula

$$\forall x \in \text{Real} : (x \neq 0) \Rightarrow (x * (3/x) = 3)$$

where *Real* is the set of all real numbers. This asserts that  $(x \neq 0) \Rightarrow (x * (3/x) = 3)$  is true for all real numbers  $x$ . Substituting 0 for  $x$  yields the true formula  $(0 \neq 0) \Rightarrow (0 * (3/0) = 3)$  that contains the silly expression  $3/0$ . It's true because  $0 \neq 0$  equals `TRUE`, and `TRUE`  $\Rightarrow P$  is true for any formula  $P$ .

A correct formula can contain silly expressions. For example,  $3/0 = 3/0$  is a correct formula because any value equals itself. However, the truth of a correct formula cannot depend on the meaning of a silly expression. If an expression is silly, then its meaning is probably unspecified. The definitions of `/` and `*` (which are in the standard module *Reals*) don't specify the value of  $0 * (3/0)$ , so there's no way of knowing whether that value equals 3.

No sensible syntactic rules can prevent you from writing  $3/0$  without also preventing you from writing perfectly reasonable expressions. The typing rules of programming languages introduce complexity and limitations on what you can write that don't exist in ordinary mathematics. In a well-designed programming language, the costs of types are balanced by benefits: types allow a compiler to produce more efficient code, and type checking catches errors. For programming languages, the benefits seem to outweigh the costs. For writing specifications, I have found that the costs outweigh the benefits.

If you're used to the constraints of programming languages, it may be a while before you start taking advantage of the freedom afforded by mathematics. At first, you won't think of defining anything like the operator *R* defined on page 50 of Section 5.2, which couldn't be written in a typed programming language.

## 6.3 Recursion Revisited

Section 5.5 introduced recursive function definitions. Let's now examine what such definitions mean mathematically. Mathematicians usually define the factorial function *fact* by writing

$$\text{fact}[n] = \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \text{fact}[n - 1], \text{ for all } n \in \text{Nat}$$

This definition can be justified by proving that it defines a unique function *fact* with domain *Nat*. In other words, *fact* is the unique value satisfying

$$(6.1) \quad \text{fact} = [n \in \text{Nat} \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \text{fact}[n - 1]]$$

The CHOOSE operator, introduced on pages 47–48 of Section 5.1, allows us to express “the value *x* satisfying property *p*” as CHOOSE *x* : *p*. We can therefore define *fact* as follows to be the value satisfying (6.1):

$$(6.2) \quad \text{fact} \triangleq \text{CHOOSE fact} :$$

$$\text{fact} = [n \in \text{Nat} \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \text{fact}[n - 1]]$$

(Since the symbol *fact* is not yet defined in the expression to the right of the “ $\triangleq$ ”, we can use it as the bound identifier in the CHOOSE expression.) The TLA<sup>+</sup> definition

$$\text{fact}[n \in \text{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \text{fact}[n - 1]$$

is simply an abbreviation for (6.2). In general, *f*[*x* ∈ *S*]  $\triangleq$  *e* is an abbreviation for

$$(6.3) \quad f \triangleq \text{CHOOSE } f : f = [x \in S \mapsto e]$$

TLA<sup>+</sup> allows you to write silly definitions. For example, you can write

$$(6.4) \quad \text{circ}[n \in \text{Nat}] \triangleq \text{CHOOSE } y : y \neq \text{circ}[n]$$

This appears to define *circ* to be a function such that *circ*[*n*]  $\neq$  *circ*[*n*] for any natural number *n*. There obviously is no such function, so *circ* can't be defined to equal it. A recursive function definition doesn't necessarily define a function. If there is no *f* that equals [*x* ∈ *S*  $\mapsto$  *e*], then (6.3) defines *f* to be some unspecified value. Thus, the nonsensical definition (6.4) defines *circ* to be some unknown value.

Although TLA<sup>+</sup> allows the apparent circularity of a recursive function definition, it does not allow circular definitions in which two or more functions are defined in terms of one another. Mathematicians occasionally write such mutually recursive definitions. For example, they might try to define functions *f* and *g*, with domains equal to the set *Nat*, by writing

$$f[n \in \text{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } f[n - 1] * g[n]$$

$$g[n \in \text{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } f[n - 1] + g[n - 1]$$

This pair of definitions is not allowed in TLA<sup>+</sup>.

TLA<sup>+</sup> does not allow mutually recursive definitions. However, we can define these functions *f* and *g* in TLA<sup>+</sup> as follows. We first define a function *mr* such that *mr*[*n*] is a record whose *f* and *g* fields equal *f*[*n*] and *g*[*n*], respectively:

$$\text{mr}[n \in \text{Nat}] \triangleq$$

$$[f \mapsto \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } \text{mr}[n - 1].f * \text{mr}[n].g,$$

$$g \mapsto \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } \text{mr}[n - 1].f + \text{mr}[n - 1].g]$$

We can then define  $f$  and  $g$  in terms of  $mr$ :

$$\begin{aligned} f[n \in Nat] &\triangleq mr[n].f \\ g[n \in Nat] &\triangleq mr[n].g \end{aligned}$$

This trick can be used to convert any mutually recursive definitions into a single recursive definition of a record-valued function whose fields are the desired functions.

If we want to reason about a function  $f$  defined by  $f[x \in S] \triangleq e$ , we need to prove that there exists an  $f$  that equals  $[x \in S \mapsto e]$ . The existence of  $f$  is obvious if  $f$  does not occur in  $e$ . If it does, so this is a recursive definition, then there is something to prove. Since I'm not discussing proofs, I won't describe how to prove it. Intuitively, you have to check that, as in the case of the factorial function, the definition uniquely determines the value of  $f[x]$  for every  $x$  in  $S$ .

Recursion is a common programming technique because programs must compute values using a small repertoire of simple elementary operations. It's not used as often in mathematical definitions, where we needn't worry about how to compute the value and can use the powerful operators of logic and set theory. For example, the operators *Head*, *Tail*, and  $\circ$  are defined in Section 5.4 without recursion, even though computer scientists usually define them recursively. Still, there are some things that are best defined inductively, using a recursive function definition.

## 6.4 Functions versus Operators

Consider these definitions, which we've seen before:

$$\begin{aligned} Tail(s) &\triangleq [i \in 1 \dots (Len(s) - 1) \mapsto s[i + 1]] \\ fact[n \in Nat] &\triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1] \end{aligned}$$

They define two very different kinds of objects:  $fact$  is a function, and  $Tail$  is an operator. Functions and operators differ in a few basic ways.

Their most obvious difference is that a function like  $fact$  by itself is a complete expression that denotes a value, but an operator like  $Tail$  is not. Both  $fact[n] \in S$  and  $fact \in S$  are syntactically correct expressions. But, while  $Tail(n) \in S$  is syntactically correct,  $Tail \in S$  is not. It is gibberish—a meaningless string of symbols, like  $x + > 0$ .

Unlike an operator, a function must have a domain, which is a set. We cannot define a function  $Tail$  so that  $Tail[s]$  is the tail of any nonempty sequence  $s$ ; the domain of such a function would have to include all nonempty sequences, and the collection of all such sequences is too big to be a set. (As explained on page 66, a collection  $\mathcal{C}$  is too big to be a set if we can assign to each set a different member of  $\mathcal{C}$ . The operator  $SMap$  defined by  $SMap(S) \triangleq \langle S \rangle$  assigns

to every set a different nonempty sequence.) Hence, we can't define *Tail* to be a function.

Unlike a function, an operator cannot be defined recursively in TLA<sup>+</sup>. However, we can usually transform an illegal recursive operator definition into a nonrecursive one using a recursive function definition. For example, let's try to define the *Cardinality* operator on finite sets. (Recall that the cardinality of a finite set  $S$  is the number of elements in  $S$ .) The collection of all finite sets is too big to be a set. (The operator  $SMap(S) \triangleq \{S\}$  assigns to each set a different set of cardinality 1.) The *Cardinality* operator has a simple intuitive definition:

- $Cardinality(\{\}) = 0$ .
- If  $S$  is a nonempty finite set, then

$$Cardinality(S) = 1 + Cardinality(S \setminus \{x\})$$

$S \setminus \{x\}$  is the set of all elements in  $S$  except  $x$ .

where  $x$  is an arbitrary element of  $S$ .

Using the CHOOSE operator to describe an arbitrary element of  $S$ , we can write this as the more formal-looking, but still illegal, definition

$$\begin{aligned} Cardinality(S) &\triangleq && \text{This is not a legal TLA}^+ \text{ definition.} \\ \text{IF } S = \{\} \text{ THEN } 0 \\ \text{ELSE } 1 + Cardinality(S \setminus \{\text{CHOOSE } x : x \in S\}) \end{aligned}$$

This definition is illegal because it's circular—only in a recursive function definition can the symbol being defined appear to the right of the  $\triangleq$ .

To turn this into a legal definition, observe that, for a given finite set  $S$ , we can define a function  $CS$  such that  $CS[T]$  equals the cardinality of  $T$  for every subset  $T$  of  $S$ . The definition is

$$\begin{aligned} CS[T \in \text{SUBSET } S] &\triangleq \\ \text{IF } T = \{\} \text{ THEN } 0 \\ \text{ELSE } 1 + CS[T \setminus \{\text{CHOOSE } x : x \in T\}] \end{aligned}$$

Since  $S$  is a subset of itself, this defines  $CS[S]$  to equal  $Cardinality(S)$ , if  $S$  is a finite set. (We don't know or care what  $CS[S]$  equals if  $S$  is not finite.) So, we can define the *Cardinality* operator by

$$\begin{aligned} Cardinality(S) &\triangleq \\ \text{LET } CS[T \in \text{SUBSET } S] &\triangleq \\ \text{IF } T = \{\} \text{ THEN } 0 \\ &\text{ELSE } 1 + CS[T \setminus \{\text{CHOOSE } x : x \in T\}] \\ \text{IN } CS[S] \end{aligned}$$

Operators also differ from functions in that an operator can take an operator as an argument. For example, we can define an operator *IsPartialOrder* so that

*IsPartialOrder*( $R, S$ ) equals true iff the operator  $R$  defines an irreflexive partial order on  $S$ . The definition is

$$\begin{aligned} \text{IsPartialOrder}(R(\_, \_), S) &\triangleq \\ &\wedge \forall x, y, z \in S : R(x, y) \wedge R(y, z) \Rightarrow R(x, z) \\ &\wedge \forall x \in S : \neg R(x, x) \end{aligned}$$

If you don't know what an irreflexive partial order is, read this definition of *IsPartialOrder* to find out.

We could also use an infix-operator symbol like  $\prec$  instead of  $R$  as the parameter of the definition, writing

$$\begin{aligned} \text{IsPartialOrder}(\_, \prec, S) &\triangleq \\ &\wedge \forall x, y, z \in S : (x \prec y) \wedge (y \prec z) \Rightarrow (x \prec z) \\ &\wedge \forall x \in S : \neg(x \prec x) \end{aligned}$$

The first argument of *IsPartialOrder* is an operator that takes two arguments; its second argument is an expression. Since  $>$  is an operator that takes two arguments, the expression *IsPartialOrder*( $>, Nat$ ) is syntactically correct. In fact, it equals TRUE, if  $>$  is defined to be the usual operator on numbers. The expression *IsPartialOrder*( $+, 3$ ) is also syntactically correct, but it's silly and we have no idea whether or not it equals TRUE.

There is one difference between functions and operators that is subtle and not very important, but I will mention it anyway for completeness. The definition of *Tail* defines *Tail*( $s$ ) for all values of  $s$ . For example, it defines *Tail*( $1/2$ ) to equal

$$(6.5) \quad [i \in 1 \dots (\text{Len}(1/2) - 1) \mapsto (1/2)[i + 1]]$$

We have no idea what this expression means, because we don't know what *Len*( $1/2$ ) or  $(1/2)[i + 1]$  mean. But, whatever (6.5) means, it equals *Tail*( $1/2$ ). The definition of *fact* defines *fact*[ $n$ ] only for  $n \in Nat$ . It tells us nothing about the value of *fact*[ $1/2$ ]. The expression *fact*[ $1/2$ ] is syntactically well-formed, so it too denotes some value. However, the definition of *fact* tells us nothing about what that value is.

The last difference between operators and functions has nothing to do with mathematics and is an idiosyncrasy of TLA<sup>+</sup>: the language doesn't permit us to define infix functions. Mathematicians often define  $/$  to be a function of two arguments, but we can't do that in TLA<sup>+</sup>. If we want to define  $/$ , we have no choice but to make it an operator.

One can write equally nonsensical things using functions or operators. However, whether you use functions or operators may determine whether the nonsense you write is nonsyntactic gibberish or syntactically correct but semantically silly. The string of symbols  $2("a")$  is not a syntactically correct formula because  $2$  is not an operator. However,  $2["a"]$ , which can also be written  $2.a$ , is a syntactically correct expression. It's nonsensical because  $2$  isn't a function,<sup>1</sup> so

<sup>1</sup>More precisely, we don't know whether or not  $2$  is a function.

we don't know what  $2["a"]$  means. Similarly,  $Tail(s, t)$  is syntactically incorrect because  $Tail$  is an operator that takes a single argument. However, as explained in Section 16.1.7 on page 301,  $fact[m, n]$  is syntactic sugar for  $fact[\langle m, n \rangle]$ , so it is a syntactically correct, semantically silly formula. Whether an error is syntactic or semantic determines what kind of tool can catch it. In particular, the parser described in Chapter 12 catches syntactic errors, but not semantic silliness. The TLC model checker, described in Chapter 14, will report an error if it tries to evaluate a semantically silly expression.

The distinction between functions and operators seems to confuse some people. One reason is that, although this distinction exists in ordinary math, it usually goes unnoticed by mathematicians. If you ask a mathematician whether  $\text{SUBSET}$  is a function, she's likely to say yes. But if you point out to her that  $\text{SUBSET}$  can't be a function because its domain can't be a set, she will probably realize for the first time that mathematicians use operators like  $\text{SUBSET}$  and  $\in$  without noticing that they form a class of objects different from functions. Logicians will observe that the distinction between operators and values, including functions, arises because  $\text{TLA}^+$  is a first-order logic rather than a higher-order logic.

When defining an object  $V$ , you may have to decide whether to make  $V$  an operator that takes an argument or a function. The differences between operators and functions will often determine the decision. For example, if a variable may have  $V$  as its value, then  $V$  must be a function. Thus, in the memory specification of Section 5.3, we had to represent the state of the memory by a function rather than an operator, since the variable  $mem$  couldn't equal an operator. If these differences don't determine whether to use an operator or a function, then the choice is a matter of taste. I usually prefer operators.

## 6.5 Using Functions

Consider the following two formulas:

$$(6.6) \quad f' = [i \in \text{Nat} \mapsto i + 1]$$

$$(6.7) \quad \forall i \in \text{Nat} : f'[i] = i + 1$$

Both formulas imply that  $f'[i] = i + 1$  for every natural number  $i$ , but they are not equivalent. Formula (6.6) uniquely determines  $f'$ , asserting that it's a function with domain  $\text{Nat}$ . Formula (6.7) is satisfied by lots of different values of  $f'$ . For example, it is satisfied if  $f'$  is the function

$$[i \in \text{Real} \mapsto \text{IF } i \in \text{Nat} \text{ THEN } i + 1 \text{ ELSE } i^2]$$

In fact, from (6.7), we can't even deduce that  $f'$  is a function. Formula (6.6) implies formula (6.7), but not vice-versa.

When writing specifications, we almost always want to specify the new value of a variable  $f$  rather than the new values of  $f[i]$  for all  $i$  in some set. We therefore usually write (6.6) rather than (6.7).

## 6.6 Choose

The CHOOSE operator was introduced in the memory interface of Section 5.1 in the simple idiom  $\text{CHOOSE } v : v \notin S$ , which is an expression whose value is not an element of  $S$ . In Section 6.3 above, we saw that it is a powerful tool that can be used in rather subtle ways.

The most common use for the CHOOSE operator is to “name” a uniquely specified value. For example,  $a/b$  is the unique real number that satisfies the formula  $a = b * (a/b)$ , if  $a$  and  $b$  are real numbers and  $b \neq 0$ . So, the standard module *Reals* defines division on the set *Real* of real numbers by

$$a/b \triangleq \text{CHOOSE } c \in \text{Real} : a = b * c$$

(The expression  $\text{CHOOSE } x \in S : p$  means  $\text{CHOOSE } x : (x \in S) \wedge p$ .) If  $a$  is a nonzero real number, then there is no real number  $c$  such that  $a = 0 * c$ . Therefore,  $a/0$  has an unspecified value. We don’t know what a real number times a string equals, so we cannot say whether or not there is a real number  $c$  such that  $a$  equals “xyz” \*  $c$ . Hence, we don’t know what the value of  $a/\text{“xyz”}$  is.

People who do a lot of programming and not much mathematics often think that CHOOSE must be a nondeterministic operator. In mathematics, there is no such thing as a nondeterministic operator or a nondeterministic function. If some expression equals 42 today, then it will equal 42 tomorrow, and it will still equal 42 a million years from tomorrow. The specification

$$(x = \text{CHOOSE } n : n \in \text{Nat}) \wedge \square[x' = \text{CHOOSE } n : n \in \text{Nat}]_x$$

allows only a single behavior—one in which  $x$  always equals  $\text{CHOOSE } n : n \in \text{Nat}$ , which is some particular, unspecified natural number. It is very different from the specification

$$(x \in \text{Nat}) \wedge \square[x' \in \text{Nat}]_x$$

that allows all behaviors in which  $x$  is always a natural number—possibly a different number in each state. This specification is highly nondeterministic, allowing lots of different behaviors.

The CHOOSE operator is known to logicians as Hilbert’s  $\varepsilon$ .



# Chapter 7

## Writing a Specification: Some Advice

You have now learned all you need to know about TLA<sup>+</sup> to write your own specifications. Here are a few additional hints to help you get started.

### 7.1 Why Specify

Writing a specification requires effort; the benefit it provides must justify that effort. The purpose of writing a specification is to help avoid errors. Here are some ways it can do that.

- Writing a TLA<sup>+</sup> specification can help the design process. Having to describe a design precisely often reveals problems—subtle interactions and “corner cases” that are easily overlooked. These problems are easier to correct when discovered in the design phase rather than after implementation has begun.
- A TLA<sup>+</sup> specification can provide a clear, concise way of communicating a design. It helps ensure that the designers agree on what they have designed, and it provides a valuable guide to the engineers who implement and test the system. It may also help users understand the system.
- A TLA<sup>+</sup> specification is a formal description to which tools can be applied to help find errors in the design and to help in testing the system. The most useful tool written so far for this purpose is the TLC model checker, described in Chapter 14.

Whether the benefit justifies the effort of writing the specification depends on the nature of the project. Specification is not an end in itself; it is just a tool that an engineer should be able to use when appropriate.

## 7.2 What to Specify

Although we talk about specifying a system, that's not what we do. A specification is a mathematical model of a particular view of some part of a system. When writing a specification, the first thing you must choose is exactly what part of the system you want to model. Sometimes the choice is obvious; often it isn't. The cache-coherence protocol of a real multiprocessor computer may be intimately connected with how the processors execute instructions. Finding an abstraction that describes the coherence protocol while suppressing the details of instruction execution may be difficult. It may require defining an interface between the processor and the memory that doesn't exist in the actual system design.

The primary purpose of a specification is to help avoid errors. You should specify those parts of the system for which a specification is most likely to reveal errors. TLA<sup>+</sup> is particularly effective at revealing concurrency errors—ones that arise through the interaction of asynchronous components. So, when writing a TLA<sup>+</sup> specification, you will probably concentrate your efforts on the parts of the system that are most likely to have such errors. If that's not where you should be concentrating your efforts, then you probably shouldn't be using TLA<sup>+</sup>.

## 7.3 The Grain of Atomicity

After choosing what part of the system to specify, you must choose the specification's level of abstraction. The most important aspect of the level of abstraction is the grain of atomicity, the choice of what system changes are represented as a single step of a behavior. Sending a message in an actual system involves multiple suboperations, but we usually represent it as a single step. On the other hand, the sending of a message and its receipt are usually represented as separate steps when specifying a distributed system.

The same sequence of system operations is represented by a shorter sequence of steps in a coarser-grained representation than in a finer-grained one. This almost always makes the coarser-grained specification simpler than the finer-grained one. However, the finer-grained specification more accurately describes the behavior of the actual system. A coarser-grained specification may fail to reveal important details of the system.

There is no simple rule for deciding on the grain of atomicity. However, there is one way of thinking about granularity that can help. To describe it, we

need the TLA<sup>+</sup> action-composition operator “.”. If  $A$  and  $B$  are actions, then the action  $A \cdot B$  is executed by executing first  $A$  then  $B$  as a single step. More precisely,  $A \cdot B$  is the action defined by letting  $s \rightarrow t$  be an  $A \cdot B$  step iff there exists a state  $u$  such that  $s \rightarrow u$  is an  $A$  step and  $u \rightarrow t$  is a  $B$  step.

When determining the grain of atomicity, we must decide whether to represent the execution of an operation as a single step or as a sequence of steps, each corresponding to the execution of a suboperation. Let’s consider the simple case of an operation consisting of two suboperations that are executed sequentially, where those suboperations are described by the two actions  $R$  and  $L$ . (Executing  $R$  enables  $L$  and disables  $R$ .) When the operation’s execution is represented by two steps, each of those steps is an  $R$  step or an  $L$  step. The operation is then described with the action  $R \vee L$ . When its execution is represented by a single step, the operation is described with the action  $R \cdot L$ .<sup>1</sup> Let  $S2$  be the finer-grained specification in which the operation is executed in two steps, and let  $S1$  be the coarser-grained specification in which it is executed as a single  $R \cdot L$  step. To choose the grain of atomicity, we must choose whether to take  $S1$  or  $S2$  as the specification. Let’s examine the relation between the two specifications.

We can transform any behavior  $\sigma$  satisfying  $S1$  into a behavior  $\hat{\sigma}$  satisfying  $S2$  by replacing each step  $s \xrightarrow{R \cdot L} t$  with the pair of steps  $s \xrightarrow{R} u \xrightarrow{L} t$ , for some state  $u$ . If we regard  $\sigma$  as being equivalent to  $\hat{\sigma}$ , then we can regard  $S1$  as being a strengthened version of  $S2$ —one that allows fewer behaviors. Specification  $S1$  requires that each  $R$  step be followed immediately by an  $L$  step, while  $S2$  allows behaviors in which other steps come between the  $R$  and  $L$  steps. To choose the appropriate grain of atomicity, we must decide whether those additional behaviors allowed by  $S2$  are important.

The additional behaviors allowed by  $S2$  are not important if the actual system executions they describe are also described by behaviors allowed by  $S1$ . So, we can ask whether each behavior  $\tau$  satisfying  $S2$  has a corresponding behavior  $\tilde{\tau}$  satisfying  $S1$  that is, in some sense, equivalent to  $\tau$ . One way to construct  $\tilde{\tau}$  from  $\tau$  is to transform a sequence of steps

$$(7.1) \quad s \xrightarrow{R} u_1 \xrightarrow{A_1} u_2 \xrightarrow{A_2} u_3 \dots u_n \xrightarrow{A_n} u_{n+1} \xrightarrow{L} t$$

into the sequence

$$(7.2) \quad s \xrightarrow{A_1} v_1 \dots v_{k-2} \xrightarrow{A_k} v_{k-1} \xrightarrow{R} v_k \xrightarrow{L} v_{k+1} \xrightarrow{A_{k+1}} v_{k+2} \dots v_{n+1} \xrightarrow{A_n} t$$

where the  $A_i$  are other system actions that can be executed between the  $R$  and  $L$  steps. Both sequences start in state  $s$  and end in state  $t$ , but the intermediate states may be different.

---

<sup>1</sup>We actually describe the operation with an ordinary action, like the ones we’ve been writing, that is equivalent to  $R \cdot L$ . The operator “.” rarely appears in an actual specification. If you’re ever tempted to use it, look for a better way to write the specification; you can probably find one.

When is such a transformation possible? An answer can be given in terms of commutativity relations. We say that actions  $A$  and  $B$  commute if performing them in either order produces the same result. Formally,  $A$  and  $B$  commute iff  $A \cdot B$  is equivalent to  $B \cdot A$ . A simple sufficient condition for commutativity is that two actions commute if (i) each one leaves unchanged any variable whose value may be changed by the other, and (ii) neither enables or disables the other. It's not hard to see that we can transform (7.1) to (7.2) in the following two cases:

- $R$  commutes with each  $A_i$ . (In this case,  $k = n$ .)
- $L$  commutes with each  $A_i$ . (In this case,  $k = 0$ .)

In general, if an operation consists of a sequence of  $m$  subactions, we must decide whether to choose the finer-grained representation  $O_1 \vee O_2 \vee \dots \vee O_m$  or the coarser-grained one  $O_1 \cdot O_2 \cdots O_m$ . The generalization of the transformation from (7.1) to (7.2) is one that transforms an arbitrary behavior satisfying the finer-grained specification into one in which the sequence of  $O_1, O_2, \dots, O_m$  steps come one right after the other. Such a transformation is possible if all but one of the actions  $O_i$  commute with every other system action. Commutativity can be replaced by weaker conditions, but it is the most common case.

By commuting actions and replacing a sequence  $s \xrightarrow{O_1} \dots \xrightarrow{O_m} t$  of steps by a single  $O_1 \cdots O_m$  step, you may be able to transform any behavior of a finer-grained specification into a corresponding behavior of a coarser-grained one. But that doesn't mean that the coarser-grained specification is just as good as the finer-grained one. The sequences (7.1) and (7.2) are not the same, and a sequence of  $O_i$  steps is not the same as a single  $O_1 \cdots O_m$  step. Whether you can consider the transformed behavior to be equivalent to the original one, and use the coarser-grained specification, depends on the particular system you are specifying and on the purpose of the specification. Understanding the relation between finer- and coarser-grained specifications can help you choose between them; it won't make the choice for you.

## 7.4 The Data Structures

Another aspect of a specification's level of abstraction is the accuracy with which it describes the system's data structures. For example, should the specification of a program interface describe the actual layout of a procedure's arguments in memory, or should the arguments be represented more abstractly?

To answer such a question, you must remember that the purpose of the specification is to help catch errors. A precise description of the layout of procedure arguments will help prevent errors caused by misunderstandings about that layout, but at the cost of complicating the program interface's specification. The

cost is justified only if such errors are likely to be a real problem and the TLA<sup>+</sup> specification provides the best way to avoid them.

If the purpose of the specification is to catch errors caused by the asynchronous interaction of concurrently executing components, then detailed descriptions of data structures will be a needless complication. So, you will probably want to use high-level, abstract descriptions of the system's data structures in the specification. For example, to specify a program interface, you might introduce constant parameters to represent the actions of calling and returning from a procedure—parameters analogous to *Send* and *Reply* of the memory interface described in Section 5.1 (page 45).

## 7.5 Writing the Specification

Once you've chosen the part of the system to specify and the level of abstraction, you're ready to start writing the TLA<sup>+</sup> specification. We've already seen how this is done; let's review the steps.

First, pick the variables and define the type invariant and initial predicate. In the course of doing this, you will determine the constant parameters and assumptions about them that you need. You may also have to define some additional constants.

Next, write the next-state action, which forms the bulk of the specification. Sketching a few sample behaviors may help you get started. You must first decide how to decompose the next-state action as the disjunction of actions describing the different kinds of system operations. You then define those actions. The goal is to make the action definitions as compact and easy to read as possible, which requires carefully structuring them. One way to reduce the size of a specification is to define state predicates and state functions that are used in several different action definitions. When writing the action definitions, you will determine which of the standard modules you need and will add the appropriate EXTENDS statement. You may also have to define some constant operators for the data structures that you are using.

You must now write the temporal part of the specification. If you want to specify liveness properties, you have to choose the fairness conditions, as described below in Chapter 8. You then combine the initial predicate, next-state action, and any fairness conditions you've chosen into the definition of a single temporal formula that is the specification.

Finally, you can assert theorems about the specification. If nothing else, you probably want to add a type-correctness theorem.

## 7.6 Some Further Hints

Here are a few miscellaneous suggestions that may help you write better specifications.

### Don't be too clever.

Cleverness can make a specification hard to read—and even wrong. The formula  $q = \langle h' \rangle \circ q'$  may look like a nice, short way of writing

$$(7.3) \quad (h' = \text{Head}(q)) \wedge (q' = \text{Tail}(q))$$

But not only is  $q = \langle h' \rangle \circ q'$  harder to understand than (7.3), it's also wrong. We don't know what  $a \circ b$  equals if  $a$  and  $b$  are not both sequences, so we don't know whether  $h' = \text{Head}(q)$  and  $q' = \text{Tail}(q)$  are the only values of  $h'$  and  $q'$  that satisfy  $q = \langle h' \rangle \circ q'$ . There could be other values of  $h'$  and  $q'$ , which are not sequences, that satisfy the formula.

In general, the best way to specify the new value of a variable  $v$  is with a conjunct of the form  $v' = \text{exp}$  or  $v' \in \text{exp}$ , where  $\text{exp}$  is a state function—an expression with no primes.

### A type invariant is not an assumption.

Type invariance is a property of a specification, not an assumption. When writing a specification, we usually define a type invariant. But that's just a definition; a definition is not an assumption. Suppose you define a type invariant that asserts that a variable  $n$  is of type  $\text{Nat}$ . You may be tempted then to think that a conjunct  $n' > 7$  in an action asserts that  $n'$  is a natural number greater than 7. It doesn't. The formula  $n' > 7$  asserts only that  $n' > 7$ . It is satisfied if  $n' = \sqrt{96}$  as well as if  $n' = 8$ . Since we don't know whether or not “abc”  $> 7$  is true, it might be satisfied even if  $n' = \text{“abc”}$ . The meaning of the formula is not changed just because you've defined a type invariant that asserts  $n \in \text{Nat}$ .

In general, you may want to describe the new value of a variable  $x$  by asserting some property of  $x'$ . However, the next-state action should imply that  $x'$  is an element of some suitable set. For example, a specification might define<sup>2</sup>

$$\begin{aligned} \text{Action1} &\triangleq (n' > 7) \wedge \dots \\ \text{Action2} &\triangleq (n' \leq 6) \wedge \dots \\ \text{Next} &\triangleq (n' \in \text{Nat}) \wedge (\text{Action1} \vee \text{Action2}) \end{aligned}$$

---

<sup>2</sup>An alternative approach is to define  $\text{Next}$  to equal  $\text{Action1} \vee \text{Action2}$  and to let the specification be  $\text{Init} \wedge \square[\text{Next}] \dots \wedge \square(n \in \text{Nat})$ . But it's usually better to stick to the simple form  $\text{Init} \wedge \square[\text{Next}] \dots$  for specifications.

### Don't be too abstract.

Suppose a user interacts with the system by typing on a keyboard. We could describe the interaction abstractly with a variable  $typ$  and an operator parameter  $KeyStroke$ , where the action  $KeyStroke("a", typ, typ')$  represents the user typing an “a”. This is the approach we took in describing the communication between the processors and the memory in the *MemoryInterface* module on page 48.

A more concrete description would be to let  $kbd$  represent the state of the keyboard, perhaps letting  $kbd = \{\}$  mean that no key is depressed, and  $kbd = \{"a"\}$  mean that the  $a$  key is depressed. The typing of an  $a$  is represented by two steps, a  $[kbd = \{\}] \rightarrow [kbd = \{"a"\}]$  step represents the pressing of the  $a$  key, and a  $[kbd = \{"a"\}] \rightarrow [kbd = \{\}]$  step represents its release. This is the approach we took in the asynchronous interface specifications of Chapter 3.

The abstract interface is simpler; typing an  $a$  is represented by a single  $KeyStroke("a", typ, typ')$  step instead of a pair of steps. However, using the concrete representation leads us naturally to ask: what if the user presses the  $a$  key and, before releasing it, presses the  $b$  key? That's easy to describe with the concrete representation. The state with both keys depressed is  $kbd = \{"a", "b"\}$ . Pressing and releasing a key are represented simply by the two actions

$$Press(k) \triangleq kbd' = kbd \cup \{k\} \quad Release(k) \triangleq kbd' = kbd \setminus \{k\}$$

The possibility of having two keys depressed cannot be expressed with the simple abstract interface. To express it abstractly, we would have to replace the parameter  $KeyStroke$  with two parameters  $PressKey$  and  $ReleaseKey$ , and we would have to express explicitly the property that a key can't be released until it has been depressed, and vice-versa. The more concrete representation is then simpler.

We might decide that we don't want to consider the possibility of two keys being depressed, and that we prefer the abstract representation. But that should be a conscious decision. Our abstraction should not blind us to what can happen in the actual system. When in doubt, it's safer to use a concrete representation that more accurately describes the real system. That way, you are less likely to overlook real problems.

### Don't assume values that look different are unequal.

The rules of TLA<sup>+</sup> do not imply that  $1 \neq "a"$ . If the system can send a message that is either a string or a number, represent the message as a record with a *type* and *value* field—for example,

$$[type \mapsto \text{String}, value \mapsto "a"] \text{ or } [type \mapsto \text{Nat}, value \mapsto 1]$$

We know that these two values are different because they have different *type* fields.

### Move quantification to the outside.

Specifications are usually easier to read if  $\exists$  is moved outside disjunctions and  $\forall$  is moved outside conjunctions. For example, instead of

$$\begin{aligned} Up &\triangleq \exists e \in Elevator : \dots \\ Down &\triangleq \exists e \in Elevator : \dots \\ Move &\triangleq Up \vee Down \end{aligned}$$

it's usually better to write

$$\begin{aligned} Up(e) &\triangleq \dots \\ Down(e) &\triangleq \dots \\ Move &\triangleq \exists e \in Elevator : Up(e) \vee Down(e) \end{aligned}$$

### Prime only what you mean to prime.

When writing an action, be careful where you put your primes. The expression  $f[e]'$  equals  $f'[e']$ ; it equals  $f'[e]$  only if  $e' = e$ , which need not be true if the expression  $e$  contains variables. Be especially careful when priming an operator whose definition contains a variable. For example, suppose  $x$  is a variable and  $op$  is defined by

$$op(a) \triangleq x + a$$

Then  $op(y)'$  equals  $(x+y)'$ , which equals  $x'+y'$ , while  $op(y')$  equals  $x+y'$ . There is no way to use  $op$  and  $'$  to write the expression  $x'+y$ . (Writing  $op'(y)$  doesn't work because it's illegal—you can prime only an expression, not an operator.)

### Write comments as comments.

Don't put comments into the specification itself. I have seen people write things like the following action definition:

$$\begin{aligned} A &\triangleq \vee \wedge x \geq 0 \\ &\quad \wedge \dots \\ &\quad \vee \wedge x < 0 \\ &\quad \wedge \text{FALSE} \end{aligned}$$

The second disjunct is meant to indicate that the writer intended  $A$  not to be enabled when  $x < 0$ . But that disjunct is completely redundant, since  $F \wedge \text{FALSE}$  equals  $\text{FALSE}$ , and  $F \vee \text{FALSE}$  equals  $F$ , for any formula  $F$ . So the second disjunct of the definition serves only as a form of comment. It's better to write

$$\begin{aligned} A &\triangleq \wedge x \geq 0 \quad A \text{ is not enabled if } x < 0 \\ &\quad \wedge \dots \end{aligned}$$

## 7.7 When and How to Specify

Specifications are often written later than they should be. Engineers are usually under severe time constraints, and they may feel that writing a specification will slow them down. Only after a design has become so complex that they need help understanding it do most engineers think about writing a precise specification.

Writing a specification helps you think clearly. Thinking clearly is hard; we can use all the help we can get. Making specification part of the design process can improve the design.

I have described how to write a specification assuming that the system design already exists. But it's better to write the specification as the system is being designed. The specification will start out being incomplete and probably incorrect. For example, an initial specification of the write-through cache of Section 5.6 (page 54) might include the definition

$RdMiss(p) \triangleq$  Enqueue a request to write value from memory to  $p$ 's cache.

Some enabling condition must be conjoined here.

$\wedge memQ' = Append(memQ, buf[p])$  Append request to  $memQ$ .

$\wedge ctl' = [ctl \text{ EXCEPT } ![p] = "?"]$  Set  $ctl[p]$  to value to be determined later.

$\wedge \text{UNCHANGED } \langle memInt, wmem, buf, cache \rangle$

Some system functionality will at first be omitted; it can be included later by adding new disjuncts to the next-state action. Tools can be applied to these preliminary specifications to help find design errors.



## Part II

# More Advanced Topics



# Chapter 8

## Liveness and Fairness

The specifications we have written so far say what a system must *not* do. The clock must not advance from 11 to 9; the receiver must not receive a message if the FIFO is empty. They don't require that the system ever actually do anything. The clock need never tick; the sender need never send any messages. Our specifications have described what are called *safety properties*. If a safety property is violated, it is violated at some particular point in the behavior—by a step that advances the clock from 11 to 9, or that reads the wrong value from memory. Therefore, we can talk about a safety property being satisfied by a finite behavior, which means that it has not been violated by any step so far.

We now learn how to specify that something *does* happen—that the clock keeps ticking, or that a value is eventually read from memory. We specify *liveness properties*—ones that cannot be violated at any particular instant. Only by examining an entire infinite behavior can we tell that the clock has stopped ticking, or that a message is never sent.

We express liveness properties as temporal formulas. This means that, to add liveness conditions to your specifications, you have to understand temporal logic—the logic of temporal formulas. The chapter begins, in Section 8.1, with a more rigorous look at what a temporal formula means. To understand a logic, you have to understand what its true formulas are. Section 8.2 is about temporal tautologies, the true formulas of temporal logic. Sections 8.4–8.7 describe how to use temporal formulas to specify liveness properties. Section 8.8 completes our study of temporal logic by examining the temporal quantifier  $\exists$ . Finally, Section 8.9 reviews what we've done and explains why the undisciplined use of temporal logic is dangerous.

This chapter is the only one that contains proofs. It would be nice if you learned to write similar proofs yourself, but it doesn't matter if you don't. The proofs are here because studying them can help you develop the intuitive understanding of temporal formulas that you need to write specifications—

an understanding that makes the truth of a simple temporal tautology like  $\square \square F \equiv \square F$  as obvious as the truth of a simple theorem about numbers like  $\forall n \in \text{Nat} : 2 * n \geq n$ .

Many readers will find that this chapter taxes their mathematical ability. Don't worry if you have trouble understanding it. Treat this chapter as an exercise to stretch your mind and prepare you to add liveness properties to your specifications. And remember that liveness properties are likely to be the least important part of your specification. You will probably not lose much if you simply omit them.

## 8.1 Temporal Formulas

Recall that a state assigns a value to every variable, and a behavior is an infinite sequence of states. A temporal formula is true or false of a behavior. Formally, a temporal formula  $F$  assigns a Boolean value, which we write  $\sigma \models F$ , to a behavior  $\sigma$ . We say that  $F$  is true of  $\sigma$ , or that  $\sigma$  satisfies  $F$ , iff  $\sigma \models F$  equals TRUE. To define the meaning of a temporal formula  $F$ , we have to explain how to determine the value of  $\sigma \models F$  for any behavior  $\sigma$ . For now, we consider only temporal formulas that don't contain the temporal existential quantifier  $\exists$ .

It's easy to define the meaning of a Boolean combination of temporal formulas in terms of the meanings of those formulas. The formula  $F \wedge G$  is true of a behavior  $\sigma$  iff both  $F$  and  $G$  are true of  $\sigma$ , and  $\neg F$  is true of  $\sigma$  iff  $F$  is not true of  $\sigma$ . These definitions are written more formally as

$$\sigma \models (F \wedge G) \triangleq (\sigma \models F) \wedge (\sigma \models G) \quad \sigma \models \neg F \triangleq \neg (\sigma \models F)$$

These are the definitions of the meaning of  $\wedge$  and of  $\neg$  as operators on temporal formulas. The meanings of the other Boolean operators are similarly defined. We can also define in this way the ordinary predicate-logic quantifiers  $\forall$  and  $\exists$  as operators on temporal formulas—for example:

$$\sigma \models (\exists r : F) \triangleq \exists r : (\sigma \models F)$$

Ordinary quantification over constant sets is defined the same way. For example, if  $S$  is an ordinary constant expression—that is, one containing no variables—then

$$\sigma \models (\forall r \in S : F) \triangleq \forall r \in S : (\sigma \models F)$$

Quantifiers are discussed further in Section 8.8 below.

All the unquantified temporal formulas that we've seen have been Boolean combinations of three simple kinds of formulas, which have the following meanings:

- A state predicate, viewed as a temporal formula, is true of a behavior iff it is true in the first state of the behavior.

*State function*  
and *state predicate* are defined  
on page 25.

- A formula  $\square P$ , where  $P$  is a state predicate, is true of a behavior iff  $P$  is true in every state of the behavior.
- A formula  $\square[N]_v$ , where  $N$  is an action and  $v$  is a state function, is true of a behavior iff every successive pair of steps in the behavior is a  $[N]_v$  step.

Since a state predicate is an action that contains no primed variables, we can both combine and generalize these three kinds of temporal formulas into the two kinds of formulas  $A$  and  $\square A$ , where  $A$  is an action. I'll first explain the meanings of these two kinds of formulas, and then define the operator  $\square$  in general. To do this, I will use the notation that  $\sigma_i$  is the  $(i + 1)^{\text{st}}$  state of the behavior  $\sigma$ , for any natural number  $i$ , so  $\sigma$  is the behavior  $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \dots$ .

We interpret an arbitrary action  $A$  as a temporal formula by defining  $\sigma \models A$  to be true iff the first two states of  $\sigma$  are an  $A$  step. That is, we define  $\sigma \models A$  to be true iff  $\sigma_0 \rightarrow \sigma_1$  is an  $A$  step. In the special case when  $A$  is a state predicate,  $\sigma_0 \rightarrow \sigma_1$  is an  $A$  step iff  $A$  is true in state  $\sigma_0$ , so this definition of  $\sigma \models A$  generalizes our interpretation of a state predicate as a temporal formula.

We have already seen that  $\square[N]_v$  is true of a behavior iff each step is a  $[N]_v$  step. This leads us to define  $\sigma \models \square A$  to be true iff  $\sigma_n \rightarrow \sigma_{n+1}$  is an  $A$  step, for all natural numbers  $n$ .

We now generalize from the definition of  $\sigma \models \square A$  for an action  $A$  to the definition of  $\sigma \models \square F$  for an arbitrary temporal formula  $F$ . We defined  $\sigma \models \square A$  to be true iff  $\sigma_n \rightarrow \sigma_{n+1}$  is an  $A$  step for all  $n$ . This is true iff  $A$ , interpreted as a temporal formula, is true of a behavior whose first step is  $\sigma_n \rightarrow \sigma_{n+1}$ , for all  $n$ . Let's define  $\sigma^{+n}$  to be the suffix of  $\sigma$  obtained by deleting its first  $n$  states:

$$\sigma^{+n} \triangleq \sigma_n \rightarrow \sigma_{n+1} \rightarrow \sigma_{n+2} \rightarrow \dots$$

Then  $\sigma_n \rightarrow \sigma_{n+1}$  is the first step of  $\sigma^{+n}$ , so  $\sigma \models \square A$  is true iff  $\sigma^{+n} \models A$  is true for all  $n$ . In other words

$$\sigma \models \square A \equiv \forall n \in \text{Nat} : \sigma^{+n} \models A$$

The obvious generalization is

$$\sigma \models \square F \triangleq \forall n \in \text{Nat} : \sigma^{+n} \models F$$

for any temporal formula  $F$ . In other words,  $\sigma$  satisfies  $\square F$  iff every suffix  $\sigma^{+n}$  of  $\sigma$  satisfies  $F$ . This defines the meaning of the temporal operator  $\square$ .

We have now defined the meaning of any temporal formula built from actions (including state predicates), Boolean operators, and the  $\square$  operator. For example:

$$\begin{aligned} \sigma \models \square((x = 1) \Rightarrow \square(y > 0)) \\ \equiv \forall n \in \text{Nat} : \sigma^{+n} \models ((x = 1) \Rightarrow \square(y > 0)) && \text{By the meaning of } \square. \\ \equiv \forall n \in \text{Nat} : (\sigma^{+n} \models (x = 1)) \Rightarrow (\sigma^{+n} \models \square(y > 0)) && \text{By the meaning of } \Rightarrow. \\ \equiv \forall n \in \text{Nat} : (\sigma^{+n} \models (x = 1)) \Rightarrow && \text{By the meaning of } \square. \\ && (\forall m \in \text{Nat} : (\sigma^{+n})^{+m} \models (y > 0)) \end{aligned}$$

Thus,  $\sigma \models \square((x = 1) \Rightarrow \square(y > 0))$  is true iff, for all  $n \in Nat$ , if  $x = 1$  is true in state  $\sigma_n$ , then  $y > 0$  is true in all states  $\sigma_{n+m}$  with  $m \geq 0$ .

To understand temporal formulas intuitively, think of  $\sigma_n$  as the state of the universe at time instant  $n$  during the behavior  $\sigma$ .<sup>1</sup> For any state predicate  $P$ , the expression  $\sigma^{+n} \models P$  asserts that  $P$  is true at time  $n$ . Thus,  $\square((x = 1) \Rightarrow \square(y > 0))$  asserts that, any time  $x = 1$  is true,  $y > 0$  is true from then on. For an arbitrary temporal formula  $F$ , we also interpret  $\sigma^{+n} \models F$  as the assertion that  $F$  is true at time instant  $n$ . The formula  $\square F$  then asserts that  $F$  is true at all times. We can therefore read  $\square$  as *always* or *henceforth* or *from then on*.

We saw in Section 2.2 that a specification should allow stuttering steps—ones that leave unchanged all the variables appearing in the formula. A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behavior should not affect the truth of the formula. We say that a formula  $F$  is *invariant under stuttering*<sup>2</sup> iff adding or deleting a stuttering step to a behavior  $\sigma$  does not affect whether  $\sigma$  satisfies  $F$ . A sensible formula should be invariant under stuttering. There's no point writing formulas that aren't sensible, so TLA allows you to write only temporal formulas that are invariant under stuttering.

A state predicate (viewed as a temporal formula) is invariant under stuttering, since its truth depends only on the first state of a behavior, and adding a stuttering step doesn't change the first state. An arbitrary action is not invariant under stuttering. For example, the action  $[x' = x + 1]_x$  is satisfied by a behavior  $\sigma$  in which  $x$  is left unchanged in the first step and incremented by 2 in the second step; it isn't satisfied by the behavior obtained by removing the initial stuttering step from  $\sigma$ . However, the formula  $\square[x' = x + 1]_x$  is invariant under stuttering, since it is satisfied by a behavior iff every step that changes  $x$  is an  $x' = x + 1$  step—a condition not affected by adding or deleting stuttering steps.

In general, the formula  $\square[A]_v$  is invariant under stuttering, for any action  $A$  and state function  $v$ . However,  $\square A$  is not invariant under stuttering for an arbitrary action  $A$ . For example,  $\square(x' = x + 1)$  can be made false by adding a step that does not change  $x$ . So, even though we have assigned a meaning to  $\square(x' = x + 1)$ , it isn't a legal TLA formula.

Invariance under stuttering is preserved by  $\square$  and by the Boolean operators—that is, if  $F$  and  $G$  are invariant under stuttering, then so are  $\square F$ ,  $\neg F$ ,  $F \wedge G$ ,  $\forall x \in S : F$ , etc. So, state predicates, formulas of the form  $\square[N]_v$ , and all formulas obtainable from them by applying  $\square$  and Boolean operators are invariant under stuttering.

<sup>1</sup>It is because we think of  $\sigma_n$  as the state at time  $n$ , and because we usually measure time starting from 0, that I number the states of a behavior starting with 0 rather than 1.

<sup>2</sup>This is a completely new sense of the word *invariant*; it has nothing to do with the concept of invariance discussed already.

We now examine five especially important classes of formulas that are constructed from arbitrary temporal formulas  $F$  and  $G$ . We introduce new operators for expressing the first three.

$\diamond F$  is defined to equal  $\neg \square \neg F$ . It asserts that  $F$  is not always false, which means that  $F$  is true at some time:

$$\begin{aligned}
 \sigma \models \diamond F & \equiv \sigma \models \neg \square \neg F & \text{By definition of } \diamond. \\
 & \equiv \neg (\sigma \models \square \neg F) & \text{By the meaning of } \neg. \\
 & \equiv \neg (\forall n \in \text{Nat} : \sigma^{+n} \models \neg F) & \text{By the meaning of } \square. \\
 & \equiv \neg (\forall n \in \text{Nat} : \neg (\sigma^{+n} \models F)) & \text{By the meaning of } \neg. \\
 & \equiv \exists n \in \text{Nat} : \sigma^{+n} \models F & \text{Because } \neg \forall \neg \text{ is equivalent to } \exists.
 \end{aligned}$$

We usually read  $\diamond$  as *eventually*, taking eventually to include now.

$F \rightsquigarrow G$  is defined to equal  $\square(F \Rightarrow \diamond G)$ . The same kind of calculation we just did for  $\sigma \models \diamond F$  shows

$$\sigma \models (F \rightsquigarrow G) \equiv \forall n \in \text{Nat} : (\sigma^{+n} \models F) \Rightarrow (\exists m \in \text{Nat} : (\sigma^{+(n+m)} \models G))$$

The formula  $F \rightsquigarrow G$  asserts that whenever  $F$  is true,  $G$  is eventually true—that is,  $G$  is true then or at some later time. We read  $\rightsquigarrow$  as *leads to*.

$\diamond \langle A \rangle_v$  is defined to equal  $\neg \square [\neg A]_v$ , where  $A$  is an action and  $v$  a state function. It asserts that not every step is a  $(\neg A) \vee (v' = v)$  step, so some step is a  $\neg((\neg A) \vee (v' = v))$  step. Since  $\neg(P \vee Q)$  is equivalent to  $(\neg P) \wedge (\neg Q)$ , for any  $P$  and  $Q$ , action  $\neg((\neg A) \vee (v' = v))$  is equivalent to  $A \wedge (v' \neq v)$ . Hence,  $\diamond \langle A \rangle_v$  asserts that some step is an  $A \wedge (v' \neq v)$  step—that is, an  $A$  step that changes  $v$ . We define the action  $\langle A \rangle_v$  by

$$\langle A \rangle_v \triangleq A \wedge (v' \neq v)$$

I pronounce  $\langle A \rangle_v$  as *angle A sub v*.

so  $\diamond \langle A \rangle_v$  asserts that eventually an  $\langle A \rangle_v$  step occurs. We think of  $\diamond \langle A \rangle_v$  as the formula obtained by applying the operator  $\diamond$  to  $\langle A \rangle_v$ , although technically it's not because  $\langle A \rangle_v$  isn't a temporal formula.

$\square \diamond F$  asserts that at all times,  $F$  is true then or at some later time. For time 0, this implies that  $F$  is true at some time  $n_0 \geq 0$ . For time  $n_0 + 1$ , it implies that  $F$  is true at some time  $n_1 \geq n_0 + 1$ . For time  $n_1 + 1$ , it implies that  $F$  is true at some time  $n_2 \geq n_1 + 1$ . Continuing the process, we see that  $F$  is true at an infinite sequence of time instants  $n_0, n_1, n_2, \dots$ . So,  $\square \diamond F$  implies that  $F$  is true at infinitely many instants. Conversely, if  $F$  is true at infinitely many instants, then, at every instant,  $F$  must be true at some later instant, so  $\square \diamond F$  is true. Therefore,  $\square \diamond F$  asserts that  $F$  is *infinitely often* true. In particular,  $\square \diamond \langle A \rangle_v$  asserts that infinitely many  $\langle A \rangle_v$  steps occur.

$\diamond\Box F$  asserts that eventually (at some time),  $F$  becomes true and remains true thereafter. In other words,  $\diamond\Box F$  asserts that  $F$  is *eventually always* true. In particular,  $\diamond\Box[N]_v$  asserts that, eventually, every step is a  $[N]_v$  step.

The operators  $\Box$  and  $\diamond$  have higher precedence (bind more tightly) than the Boolean operators, so  $\diamond F \vee \Box G$  means  $(\diamond F) \vee (\Box G)$ . The operator  $\rightsquigarrow$  has lower precedence than  $\wedge$  and  $\vee$ .

## 8.2 Temporal Tautologies

A temporal theorem is a temporal formula that is satisfied by all behaviors. In other words,  $F$  is a theorem iff  $\sigma \models F$  equals TRUE for all behaviors  $\sigma$ . For example, the *HourClock* module asserts that  $HC \Rightarrow \Box HCini$  is a theorem, where  $HC$  and  $HCini$  are the formulas defined in the module. This theorem expresses a property of the hour clock.

The formula  $\Box HCini \Rightarrow HCini$  is also a theorem. However, it tells us nothing about the hour clock because it's true regardless of how  $HCini$  is defined. For example, substituting  $x > 7$  for  $HCini$  yields the theorem  $\Box(x > 7) \Rightarrow (x > 7)$ . A formula like  $\Box HCini \Rightarrow HCini$  that is true when any formulas are substituted for its identifiers is called a *tautology*. To distinguish them from the tautologies of ordinary logic, tautologies containing temporal operators are sometimes called *temporal tautologies*.

Let's prove that  $\Box HCini \Rightarrow HCini$  is a temporal tautology. To avoid confusing the arbitrary identifier  $HCini$  in this tautology with the formula  $HCini$  defined in the *HourClock* module, let's replace it by  $F$ , so the tautology becomes  $\Box F \Rightarrow F$ . There are axioms and inference rules for temporal logic from which we can prove any temporal tautology that, like  $\Box F \Rightarrow F$ , contains no quantifiers. However, it's often easier and more instructive to prove them directly from the meanings of the operators. We prove that  $\Box F \Rightarrow F$  is a tautology by proving that  $\sigma \models (\Box F \Rightarrow F)$  equals TRUE, for any behavior  $\sigma$  and any formula  $F$ . The proof is simple:

$$\begin{aligned}
 \sigma \models (\Box F \Rightarrow F) &\equiv (\sigma \models \Box F) \Rightarrow (\sigma \models F) && \text{By the meaning of } \Rightarrow. \\
 &\equiv (\forall n \in \text{Nat} : \sigma^{+n} \models F) \Rightarrow (\sigma \models F) && \text{By definition of } \Box. \\
 &\equiv (\forall n \in \text{Nat} : \sigma^{+n} \models F) \Rightarrow (\sigma^{+0} \models F) && \text{By definition of } \sigma^{+0}. \\
 &\equiv \text{TRUE} && \text{By predicate logic.}
 \end{aligned}$$

The temporal tautology  $\Box F \Rightarrow F$  asserts the obvious fact that, if  $F$  is true at all times, then it's true at time 0. Such a simple tautology should be obvious once you become accustomed to thinking in terms of temporal formulas. Here are three more simple tautologies, along with their English translations.

$$\neg\Box F \equiv \diamond\neg F$$

$F$  is not always true iff it is eventually false.

$$\square(F \wedge G) \equiv (\square F) \wedge (\square G)$$

$F$  and  $G$  are both always true iff  $F$  is always true and  $G$  is always true.

Another way of saying this is that  $\square$  distributes over  $\wedge$ .

$$\diamond(F \vee G) \equiv (\diamond F) \vee (\diamond G)$$

$F$  or  $G$  is eventually true iff  $F$  is eventually true or  $G$  is eventually true.

Another way of saying this is that  $\diamond$  distributes over  $\vee$ .

At the heart of the proof of each of these tautologies is a tautology of predicate logic. For example, the proof that  $\square$  distributes over  $\wedge$  relies on the fact that  $\forall$  distributes over  $\wedge$ :

$$\begin{aligned} \sigma \models (\square(F \wedge G) \equiv (\square F) \wedge (\square G)) \\ \equiv (\sigma \models \square(F \wedge G)) \equiv (\sigma \models (\square F) \wedge (\square G)) & \quad \text{By the meaning of } \equiv. \\ \equiv (\sigma \models \square(F \wedge G)) \equiv (\sigma \models \square F) \wedge (\sigma \models \square G) & \quad \text{By the meaning of } \wedge. \\ \equiv (\forall n \in \text{Nat} : \sigma^{+n} \models (F \wedge G)) \equiv & \quad \text{By definition of } \square. \\ (\forall n \in \text{Nat} : \sigma^{+n} \models F) \wedge (\forall n \in \text{Nat} : \sigma^{+n} \models G) \\ \equiv \text{TRUE} & \quad \text{By the predicate-logic tautology } (\forall x \in S : P \wedge Q) \equiv (\forall x \in S : P) \wedge (\forall x \in S : Q). \end{aligned}$$

The operator  $\square$  doesn't distribute over  $\vee$ , nor does  $\diamond$  distribute over  $\wedge$ . For example,  $\square((n \geq 0) \vee (n < 0))$  is not equivalent to  $(\square(n \geq 0) \vee \square(n < 0))$ ; the first formula is true for any behavior in which  $n$  is always a number, but the second is false for a behavior in which  $n$  assumes both positive and negative values. However, the following two formulas are tautologies:

$$(\square F) \vee (\square G) \Rightarrow \square(F \vee G) \quad \diamond(F \wedge G) \Rightarrow (\diamond F) \wedge (\diamond G)$$

Either of these tautologies can be derived from the other by substituting  $\neg F$  for  $F$  and  $\neg G$  for  $G$ . Making this substitution in the second tautology yields

$$\begin{aligned} \text{TRUE} \equiv \diamond((\neg F) \wedge (\neg G)) \Rightarrow (\diamond \neg F) \wedge (\diamond \neg G) & \quad \text{By substitution in the second tautology.} \\ \equiv \diamond \neg(F \vee G) \Rightarrow (\diamond \neg F) \wedge (\diamond \neg G) & \quad \text{Because } (\neg P \wedge \neg Q) \equiv \neg(P \vee Q). \\ \equiv \neg \square(F \vee G) \Rightarrow (\neg \square F) \wedge (\neg \square G) & \quad \text{Because } \diamond \neg H \equiv \neg \square H. \\ \equiv \neg \square(F \vee G) \Rightarrow \neg((\square F) \vee (\square G)) & \quad \text{Because } (\neg P \wedge \neg Q) \equiv \neg(P \vee Q). \\ \equiv (\square F) \vee (\square G) \Rightarrow \square(F \vee G) & \quad \text{Because } (\neg P \Rightarrow \neg Q) \equiv (Q \Rightarrow P). \end{aligned}$$

This pair of tautologies illustrates a general law: from any temporal tautology, we obtain a *dual* tautology by making the replacements

$$\square \leftarrow \diamond \quad \diamond \leftarrow \square \quad \wedge \leftarrow \vee \quad \vee \leftarrow \wedge$$

and reversing the direction of all implications. (Any  $\equiv$  or  $\neg$  is left unchanged.) As in the example above, the dual tautology can be proved from the original by replacing each identifier with its negation and applying the (dual) tautologies  $\diamond \neg F \equiv \neg \square F$  and  $\neg \diamond F \equiv \square \neg F$  along with propositional-logic reasoning.

Another important pair of dual tautologies assert that  $\square\Diamond$  distributes over  $\vee$  and  $\Diamond\square$  distributes over  $\wedge$ :

$$(8.1) \quad \square\Diamond(F \vee G) \equiv (\square\Diamond F) \vee (\square\Diamond G) \quad \Diamond\square(F \wedge G) \equiv (\Diamond\square F) \wedge (\Diamond\square G)$$

The first asserts that  $F$  or  $G$  is true infinitely often iff  $F$  is true infinitely often or  $G$  is true infinitely often. Its truth should be fairly obvious, but let's prove it. To reason about  $\square\Diamond$ , it helps to introduce the symbol  $\exists_\infty$ , which means *there exist infinitely many*. In particular,  $\exists_\infty i \in \text{Nat} : P(i)$  means that  $P(i)$  is true for infinitely many natural numbers  $i$ . On page 91, we showed that  $\square\Diamond F$  asserts that  $F$  is true infinitely often. Using  $\exists_\infty$ , we can express this as

$$(8.2) \quad (\sigma \models \square\Diamond F) \equiv (\exists_\infty i \in \text{Nat} : \sigma^{+i} \models F)$$

The same reasoning proves the following more general result, where  $P$  is any operator:

$$(8.3) \quad (\forall n \in \text{Nat} : \exists m \in \text{Nat} : P(n + m)) \equiv \exists_\infty i \in \text{Nat} : P(i)$$

Here is another useful tautology involving  $\exists_\infty$ , where  $P$  and  $Q$  are arbitrary operators and  $S$  is an arbitrary set:

$$(8.4) \quad (\exists_\infty i \in S : P(i) \vee Q(i)) \equiv (\exists_\infty i \in S : P(i)) \vee (\exists_\infty i \in S : Q(i))$$

Using these results, it's now easy to prove that  $\square\Diamond$  distributes over  $\vee$ :

$$\begin{aligned} \sigma \models \square\Diamond(F \vee G) \\ \equiv \exists_\infty i \in \text{Nat} : \sigma^{+i} \models (F \vee G) & \quad \text{By (8.2).} \\ \equiv (\exists_\infty i \in \text{Nat} : \sigma^{+i} \models F) \vee (\exists_\infty i \in \text{Nat} : \sigma^{+i} \models G) & \quad \text{By (8.4).} \\ \equiv (\sigma \models \square\Diamond F) \vee (\sigma \models \square\Diamond G) & \quad \text{By (8.2).} \end{aligned}$$

From this, we deduce the dual tautology, that  $\Diamond\square$  distributes over  $\wedge$ .

In any TLA tautology, replacing a temporal formula by an action yields a tautology—a formula that is true for all behaviors—even if that formula isn't a legal TLA formula. (Remember that we have defined the meaning of nonTLA formulas like  $\square(x' = x + 1)$ .) We can apply the rules of logic to transform those nonTLA tautologies into TLA tautologies. Among these rules are the following dual equivalences, which are easy to check:

$$[A \wedge B]_v \equiv [A]_v \wedge [B]_v \quad \langle A \vee B \rangle_v \equiv \langle A \rangle_v \vee \langle B \rangle_v$$

(The second asserts that an  $A \vee B$  step that changes  $v$  is either an  $A$  step that changes  $v$  or a  $B$  step that changes  $v$ .)

As an example of substituting actions for temporal formulas in TLA tautologies, let's substitute  $\langle A \rangle_v$  and  $\langle B \rangle_v$  for  $F$  and  $G$  in the first tautology of (8.1) to get

$$(8.5) \quad \square\Diamond(\langle A \rangle_v \vee \langle B \rangle_v) \equiv (\square\Diamond\langle A \rangle_v) \vee (\square\Diamond\langle B \rangle_v)$$

This isn't a TLA tautology, because  $\square\lozenge(\langle A \rangle_v \vee \langle B \rangle_v)$  isn't a TLA formula. However, a general rule of logic tells us that replacing a subformula by an equivalent one yields an equivalent formula. Substituting  $\langle A \vee B \rangle_v$  for  $\langle A \rangle_v \vee \langle B \rangle_v$  in (8.5) gives us the following TLA tautology:

$$\square\lozenge\langle A \vee B \rangle_v \equiv (\square\lozenge\langle A \rangle_v) \vee (\square\lozenge\langle B \rangle_v)$$

## 8.3 Temporal Proof Rules

A proof rule is a rule for deducing true formulas from other true formulas. For example, the *Modus Ponens* Rule of propositional logic tells us that, for any formulas  $F$  and  $G$ , if we have proved  $F$  and  $F \Rightarrow G$ , then we can deduce  $G$ . Since the laws of propositional logic hold for temporal logic as well, we can apply the *Modus Ponens* Rule when reasoning about temporal formulas. Temporal logic also has some proof rules of its own. One is

**Generalization Rule** From  $F$  we can infer  $\square F$ , for any temporal formula  $F$ .

This rule asserts that, if  $F$  is true for all behaviors, then so is  $\square F$ . To prove it, we must show that, if  $\sigma \models F$  is true for every behavior  $\sigma$ , then  $\tau \models \square F$  is true for every behavior  $\tau$ . The proof is easy:

$$\begin{aligned} \tau \models \square F &\equiv \forall n \in \text{Nat} : \tau^{+n} \models F && \text{By definition of } \square. \\ &\equiv \forall n \in \text{Nat} : \text{TRUE} && \text{By the assumption that } \sigma \models F \text{ equals TRUE, for all } \sigma. \\ &\equiv \text{TRUE} && \text{By predicate logic.} \end{aligned}$$

Another temporal proof rule is

**Implies Generalization Rule** From  $F \Rightarrow G$  we can infer  $\square F \Rightarrow \square G$ , for any temporal formulas  $F$  and  $G$ .

The Generalization Rule can be derived from the Implies Generalization Rule and the tautology  $\text{TRUE} = \square \text{TRUE}$  by substituting  $\text{TRUE}$  for  $F$  and  $F$  for  $G$ .

The difference between a temporal proof rule and a temporal tautology can be confusing. In propositional logic, every proof rule has a corresponding tautology. The *Modus Ponens* Rule, which asserts that we can deduce  $G$  by proving  $F$  and  $F \Rightarrow G$ , implies the tautology  $F \wedge (F \Rightarrow G) \Rightarrow G$ . But in temporal logic, a proof rule need not imply a tautology. The Generalization Rule, which states that we can deduce  $\square F$  by proving  $F$ , does not imply that  $F \Rightarrow \square F$  is a tautology. The rule means that, if  $\sigma \models F$  is true for all  $\sigma$ , then  $\sigma \models \square F$  is true for all  $\sigma$ . That's different from the (false) assertion that  $F \Rightarrow \square F$  is a tautology, which would mean that  $\sigma \models (F \Rightarrow \square F)$  is true for all  $\sigma$ . For example,  $\sigma \models (F \Rightarrow \square F)$  equals  $\text{FALSE}$  if  $F$  is a state predicate that is true in the first state of  $\sigma$  and is false in some other state of  $\sigma$ . Forgetting the distinction between a proof rule and a tautology is a common source of mistakes when using temporal logic.

## 8.4 Weak Fairness

It's easy to specify liveness properties with the temporal operators  $\square$  and  $\diamond$ . For example, consider the hour-clock specification of module *HourClock* in Figure 2.1 on page 20. We can require that the clock never stops by asserting that there must be infinitely many *HCnxt* steps. The obvious way to write this assertion is  $\square\diamond HCnxt$ , but that's not a legal TLA formula because *HCnxt* is an action, not a temporal formula. However, an *HCnxt* step advances the value *hr* of the clock, so it changes *hr*. Therefore, an *HCnxt* step is also an *HCnxt* step that changes *hr*—that is, it's an  $\langle HCnxt \rangle_{hr}$  step. We can thus write the liveness property that the clock never stops as  $\square\diamond\langle HCnxt \rangle_{hr}$ . So, we can take  $HC \wedge \square\diamond\langle HCnxt \rangle_{hr}$  to be the specification of a clock that never stops.

Before continuing, I must make a confession and then lead you on a brief digression about subscripts. Let me first confess that the argument I just gave, that we can write  $\square\diamond\langle HCnxt \rangle_{hr}$  in place of  $\square\diamond HCnxt$ , was sloppy (a polite term for *wrong*). Not every *HCnxt* step changes *hr*. Consider a state in which *hr* has some value that is not a number—perhaps a value  $\infty$ . An *HCnxt* step that starts in such a state sets the new value of *hr* to  $\infty + 1$ . We don't know what  $\infty + 1$  equals; it might or might not equal  $\infty$ . If it does, then the *HCnxt* step leaves *hr* unchanged, so it is not an  $\langle HCnxt \rangle_{hr}$  step. Fortunately, states in which the value of *hr* is not a number are irrelevant. Because we are conjoining the liveness condition to the safety specification *HC*, we care only about behaviors that satisfy *HC*. In all such behaviors, *hr* is always a number, and every *HCnxt* step is an  $\langle HCnxt \rangle_{hr}$  step. Therefore,  $HC \wedge \square\diamond\langle HCnxt \rangle_{hr}$  is equivalent to the nonTLA formula  $HC \wedge \square\diamond HCnxt$ .<sup>3</sup>

When writing liveness properties, the syntax of TLA often forces us to write  $\langle A \rangle_v$  instead of *A*, for some action *A*. As in the case of *HCnxt*, the safety specification usually implies that any *A* step changes some variable. To avoid having to think about which variables *A* actually changes, we generally take the subscript *v* to be the tuple of all variables, which is changed iff any variable changes. But what if *A* does allow stuttering steps? It's silly to assert that a stuttering step eventually occurs, since such an assertion is not invariant under stuttering. So, if *A* does allow stuttering steps, we want to require not that an *A* step eventually occurs, but that a nonstuttering *A* step occurs—that is, an  $\langle A \rangle_v$  step, where *v* is the tuple of all the specification's variables. The syntax of TLA forces us to say what we should mean.

When discussing formulas, I will usually ignore the angle brackets and subscripts. For example, I might describe  $\square\diamond\langle HCnxt \rangle_{hr}$  as the assertion that there are infinitely many *HCnxt* steps, rather than infinitely many  $\langle Hnxt \rangle_{hr}$ , which is what it really asserts. This finishes the digression; we now return to specifying liveness conditions.

---

<sup>3</sup>Even though  $HC \wedge \square\diamond HCnxt$  is not a TLA formula, its meaning has been defined, so we can determine whether it is equivalent to a TLA formula.

Let's modify specification *Spec* of module *Channel* (Figure 3.2 on page 30) to require that every value sent is eventually received. We do this by conjoining a liveness condition to *Spec*. The analog of the liveness condition for the clock is  $\square\Diamond\langle Rcv \rangle_{chan}$ , which asserts that there are infinitely many *Rcv* steps. However, only a value that has been sent can be received, so this condition would also require that infinitely many values be sent—a requirement we don't want to make. We want to permit behaviors in which no value is ever sent, so no value is ever received. We require only that any value that is sent is eventually received.

To assure that all values that should be received are eventually received, it suffices to require only that the next value to be received eventually is received. (When that value has been received, the one after it becomes the next value to be received, so it must eventually be received, and so on.) More precisely, we need only require it always to be the case that, if there is a value to be received, then the next value to be received eventually is received. The next value is received by a *Rcv* step, so the requirement is<sup>4</sup>

$$\square(\text{There is an unreceived value} \Rightarrow \Diamond\langle Rcv \rangle_{chan})$$

There is an unreceived value iff action *Rcv* is enabled, meaning that it is possible to take a *Rcv* step. TLA<sup>+</sup> defines *ENABLED A* to be the predicate that is true iff action *A* is enabled. The liveness condition can then be written

$$(8.6) \quad \square(\text{ENABLED } \langle Rcv \rangle_{chan} \Rightarrow \Diamond\langle Rcv \rangle_{chan})$$

In the *ENABLED* formula, it doesn't matter if we write *Rcv* or  $\langle Rcv \rangle_{chan}$ . We add the angle brackets so the two actions appearing in the formula are the same.

In any behavior satisfying the safety specification *HC*, it's always possible to take an *HCnxt* step that changes *hr*. Action  $\langle HCnxt \rangle_{hr}$  is therefore always enabled, so *ENABLED*  $\langle HCnxt \rangle_{hr}$  is true throughout such a behavior. Since  $\text{TRUE} \Rightarrow \Diamond\langle HCnxt \rangle_{hr}$  is equivalent to  $\Diamond\langle HCnxt \rangle_{hr}$ , we can replace the liveness condition  $\square\Diamond\langle HCnxt \rangle_{hr}$  for the hour clock with

$$\square(\text{ENABLED } \langle HCnxt \rangle_{hr} \Rightarrow \Diamond\langle HCnxt \rangle_{hr})$$

This suggests the following general liveness condition for an action *A*:

$$\square(\text{ENABLED } \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$$

This condition asserts that, if *A* ever becomes enabled, then an *A* step will eventually occur—even if *A* remains enabled for only a fraction of a nanosecond and is never again enabled. The obvious practical difficulty of implementing such a condition suggests that it's too strong. So, we replace it with the weaker formula *WF*<sub>v</sub>(*A*), defined to equal

$$(8.7) \quad \square(\square\text{ENABLED } \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v)$$

---

<sup>4</sup> $\square(F \Rightarrow \Diamond G)$  equals  $F \rightsquigarrow G$ , so we could write this formula more compactly with  $\rightsquigarrow$ . However, it's more convenient to keep it in the form  $\square(F \Rightarrow \Diamond G)$

This formula asserts that, if  $A$  ever becomes forever enabled, then an  $A$  step must eventually occur. WF stands for *Weak Fairness*, and the condition  $\text{WF}_v(A)$  is called *weak fairness on  $A$* . We'll soon see that our liveness conditions for the clock and the channel can be written as WF formulas. But first, let's examine (8.7) and the following two formulas, which turn out to be equivalent to it:

$$(8.8) \quad \square\Diamond(\neg\text{ENABLED } \langle A \rangle_v) \vee \square\Diamond\langle A \rangle_v$$

$$(8.9) \quad \Diamond\square(\text{ENABLED } \langle A \rangle_v) \Rightarrow \square\Diamond\langle A \rangle_v$$

These three formulas can be expressed in English as

(8.7) It's always the case that, if  $A$  is enabled forever, then an  $A$  step eventually occurs.

(8.8)  $A$  is infinitely often disabled, or infinitely many  $A$  steps occur.

(8.9) If  $A$  is eventually enabled forever, then infinitely many  $A$  steps occur.

The equivalence of these three formulas isn't obvious. Trying to deduce their equivalence from the English expressions often leads to confusion. The best way to avoid confusion is to use mathematics. We show that the three formulas are equivalent by proving that (8.7) is equivalent to (8.8) and that (8.8) is equivalent to (8.9). Instead of proving that they are equivalent for an individual behavior, we can use tautologies that we've already seen to prove their equivalence directly. Here's a proof that (8.7) is equivalent to (8.8). Studying it will help you learn to write liveness conditions.

$$\begin{aligned}
 & \square(\square\text{ENABLED } \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v) \\
 & \equiv \square(\neg\square\text{ENABLED } \langle A \rangle_v \vee \Diamond\langle A \rangle_v) && \text{Because } (F \Rightarrow G) \equiv (\neg F \vee G). \\
 & \equiv \square(\Diamond\neg\text{ENABLED } \langle A \rangle_v \vee \Diamond\langle A \rangle_v) && \text{Because } \neg\square F \equiv \Diamond\neg F. \\
 & \equiv \square\Diamond(\neg\text{ENABLED } \langle A \rangle_v \vee \langle A \rangle_v) && \text{Because } \Diamond F \vee \Diamond G \equiv \Diamond(F \vee G). \\
 & \equiv \square\Diamond(\neg\text{ENABLED } \langle A \rangle_v) \vee \square\Diamond\langle A \rangle_v && \text{Because } \square\Diamond(F \vee G) \equiv \square\Diamond F \vee \square\Diamond G.
 \end{aligned}$$

The equivalence of (8.8) and (8.9) is proved as follows:

$$\begin{aligned}
 & \square\Diamond(\neg\text{ENABLED } \langle A \rangle_v) \vee \square\Diamond\langle A \rangle_v \\
 & \equiv \neg\Diamond\square(\text{ENABLED } \langle A \rangle_v) \vee \square\Diamond\langle A \rangle_v && \text{Because } \square\Diamond\neg F \equiv \neg\square\Diamond F. \\
 & \equiv \Diamond\square(\text{ENABLED } \langle A \rangle_v) \Rightarrow \square\Diamond\langle A \rangle_v && \text{Because } (F \Rightarrow G) \equiv (\neg F \vee G).
 \end{aligned}$$

We now show that the liveness conditions for the hour clock and the channel can be written as weak fairness conditions.

First, consider the hour clock. In any behavior satisfying  $HC$ , an  $\langle HC_{nxt} \rangle_{hr}$  step is always enabled, so  $\Diamond\square(\text{ENABLED } \langle HC_{nxt} \rangle_{hr})$  equals TRUE. Therefore,  $HC$  implies that  $\text{WF}_{hr}(HC_{nxt})$ , which equals (8.9), is equivalent to formula  $\square\Diamond\langle HC_{nxt} \rangle_{hr}$ , our liveness condition for the hour clock.

Now, consider the channel. I claim that the liveness condition (8.6) can be replaced by  $\text{WF}_{\text{chan}}(\text{Rcv})$ . More precisely,  $\text{Spec}$  implies that these two formulas are equivalent, so conjoining either of them to  $\text{Spec}$  yields equivalent specifications. The proof rests on the observation that, in any behavior satisfying  $\text{Spec}$ , once  $\text{Rcv}$  becomes enabled (because a value has been sent), it can be disabled only by a  $\text{Rcv}$  step (which receives the value). In other words, it's always the case that if  $\text{Rcv}$  is enabled, then it is enabled forever or a  $\text{Rcv}$  step eventually occurs. Stated formally, this observation asserts that  $\text{Spec}$  implies

$$(8.10) \quad \square(\text{ENABLED } \langle \text{Rcv} \rangle_{\text{chan}} \Rightarrow \square(\text{ENABLED } \langle \text{Rcv} \rangle_{\text{chan}}) \vee \diamond \langle \text{Rcv} \rangle_{\text{chan}})$$

We show that we can take  $\text{WF}_{\text{chan}}(\text{Rcv})$  as our liveness condition by showing that (8.10) implies the equivalence of (8.6) and  $\text{WF}_{\text{chan}}(\text{Rcv})$ .

The proof is by purely temporal reasoning; we need no other facts about the channel specification. Both for compactness and to emphasize the generality of our reasoning, let's replace  $\text{ENABLED } \langle \text{Rcv} \rangle_{\text{chan}}$  by  $E$  and  $\langle \text{Rcv} \rangle_{\text{chan}}$  by  $A$ . Using version (8.7) of the definition of  $\text{WF}$ , we must prove

$$(8.11) \quad \square(E \Rightarrow \square E \vee \diamond A) \Rightarrow (\square(E \Rightarrow \diamond A) \equiv \square(\square E \Rightarrow \diamond A))$$

So far, all our proofs have been by calculation. That is, we have proved that two formulas are equivalent, or that a formula is equivalent to  $\text{TRUE}$ , by proving a chain of equivalences. That's a good way to prove simple things, but it's usually better to tackle a complicated formula like (8.11) by splitting its proof into pieces. We have to prove that one formula implies the equivalence of two others. The equivalence of two formulas can be proved by showing that each implies the other. More generally, to prove that  $P$  implies  $Q \equiv R$ , we prove that  $P \wedge Q$  implies  $R$  and that  $P \wedge R$  implies  $Q$ . So, we prove (8.11) by proving the two formulas

$$(8.12) \quad \square(E \Rightarrow \square E \vee \diamond A) \wedge \square(E \Rightarrow \diamond A) \Rightarrow \square(\square E \Rightarrow \diamond A)$$

$$(8.13) \quad \square(E \Rightarrow \square E \vee \diamond A) \wedge \square(\square E \Rightarrow \diamond A) \Rightarrow \square(E \Rightarrow \diamond A)$$

Both (8.12) and (8.13) have the form  $\square F \wedge \square G \Rightarrow \square H$ . We first show that, for any formulas  $F$ ,  $G$ , and  $H$ , we can deduce  $\square F \wedge \square G \Rightarrow \square H$  by proving  $F \wedge G \Rightarrow H$ . We do this by assuming  $F \wedge G \Rightarrow H$  and proving  $\square F \wedge \square G \Rightarrow \square H$  as follows:

1.  $\square(F \wedge G) \Rightarrow \square H$

PROOF: By the assumption  $F \wedge G \Rightarrow H$  and the Implies Generalization Rule (page 95), substituting  $F \wedge G$  for  $F$  and  $H$  for  $G$  in the rule.

2.  $\square F \wedge \square G \Rightarrow \square H$

PROOF: By step 1 and the tautology  $\square(F \wedge G) \equiv \square F \wedge \square G$ .

This shows that we can deduce  $\square F \wedge \square G \Rightarrow \square H$  by proving  $F \wedge G \Rightarrow H$ , for any  $F$ ,  $G$ , and  $H$ . We can therefore prove (8.12) and (8.13) by proving

$$(8.14) (E \Rightarrow \square E \vee \diamond A) \wedge (E \Rightarrow \diamond A) \Rightarrow (\square E \Rightarrow \diamond A)$$

$$(8.15) (E \Rightarrow \square E \vee \diamond A) \wedge (\square E \Rightarrow \diamond A) \Rightarrow (E \Rightarrow \diamond A)$$

The proof of (8.14) is easy. In fact, we don't even need the first conjunct; we can prove  $(E \Rightarrow \diamond A) \Rightarrow (\square E \Rightarrow \diamond A)$  as follows:

$$(E \Rightarrow \diamond A)$$

$\equiv (\square E \Rightarrow E) \wedge (E \Rightarrow \diamond A)$  Because  $\square E \Rightarrow E$  is a temporal tautology.

$\Rightarrow (\square E \Rightarrow \diamond A)$  By the tautology  $(P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$ .

The proof of (8.15) uses only propositional logic. We deduce (8.15) by substituting  $E$  for  $P$ ,  $\square E$  for  $Q$ , and  $\diamond A$  for  $R$  in the following propositional-logic tautology:

$$(P \Rightarrow Q \vee R) \wedge (Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$$

A little thought should make the validity of this tautology seem obvious. If not, you can check it by constructing a truth table.

These proofs of (8.14) and (8.15) complete the proof that we can take  $\text{WF}_{\text{chan}}(\text{Rcv})$  instead of (8.7) as our liveness condition for the channel.

## 8.5 The Memory Specification

### 8.5.1 The Liveness Requirement

Let's now strengthen the specification of the linearizable memory of Section 5.3 with the liveness requirement that every request must receive a response. (We don't require that a request ever be issued.) The liveness requirement is conjoined to the internal memory specification, formula  $ISpec$  of the *InternalMemory* module (Figure 5.2 on pages 52–53).

We want to express the liveness requirement in terms of weak fairness. To do this, we must understand when actions are enabled. The action  $Rsp(p)$  is enabled only if the action

$$(8.16) \text{Reply}(p, \text{buf}[p], \text{memInt}, \text{memInt}')$$

is enabled. Recall that the operator *Reply* is a constant parameter, declared in the *MemoryInterface* module (Figure 5.1 on page 48). Without knowing more about this operator, we can't say when action (8.16) is enabled.

Let's assume that *Reply* actions are always enabled. That is, for any processor  $p$  and reply  $r$ , and any old value  $miOld$  of *memInt*, there is a new value

$miNew$  of  $memInt$  such that  $Reply(p, r, miOld, miNew)$  is true. For simplicity, we just assume that this is true for all  $p$  and  $r$ , and add the following assumption to the *MemoryInterface* module:

$$\text{ASSUME } \forall p, r, miOld : \exists miNew : Reply(p, r, miOld, miNew)$$

We should also make a similar assumption for *Send*, but we don't need it here.

We will subscript our weak fairness formulas with the tuple of all variables, so let's give that tuple a name:

$$vars \triangleq \langle memInt, mem, ctl, buf \rangle$$

When processor  $p$  issues a request, it enables the  $Do(p)$  action, which remains enabled until a  $Do(p)$  step occurs. The weak fairness condition  $\text{WF}_{vars}(Do(p))$  implies that this  $Do(p)$  step must eventually occur. A  $Do(p)$  step enables the  $Rsp(p)$  action, which remains enabled until a  $Rsp(p)$  step occurs. The weak fairness condition  $\text{WF}_{vars}(Rsp(p))$  implies that this  $Rsp(p)$  step, which produces the desired response, must eventually occur. Hence, the requirement

$$(8.17) \text{ WF}_{vars}(Do(p)) \wedge \text{WF}_{vars}(Rsp(p))$$

assures that every request issued by processor  $p$  must eventually receive a reply. We want this condition to hold for every processor  $p$ , so we can take, as the liveness condition for the memory specification, the formula

$$(8.18) \text{ Liveness} \triangleq \forall p \in Proc : \text{WF}_{vars}(Do(p)) \wedge \text{WF}_{vars}(Rsp(p))$$

The internal memory specification is then  $ISpec \wedge \text{Liveness}$ .

### 8.5.2 Another Way to Write It

I find a single fairness condition simpler than the conjunction of fairness conditions. Seeing the conjunction of the two weak fairness formulas in the definition of *Liveness* leads me to ask if it can be replaced by a single weak fairness condition on  $Do(p) \vee Rsp(p)$ . Such a replacement isn't always possible; in general, the formulas  $\text{WF}_v(A) \wedge \text{WF}_v(B)$  and  $\text{WF}_v(A \vee B)$  are not equivalent. However, in this case, we can replace the two fairness conditions with one. If we define

$$(8.19) \text{ Liveness2} \triangleq \forall p \in Proc : \text{WF}_{vars}(Do(p) \vee Rsp(p))$$

then  $ISpec \wedge \text{Liveness2}$  is equivalent to  $ISpec \wedge \text{Liveness}$ . As we will see, this equivalence holds because any behavior satisfying *ISpec* satisfies the following two properties:

- DR1. Whenever  $Do(p)$  is enabled,  $Rsp(p)$  can never become enabled unless a  $Do(p)$  step eventually occurs.

- DR2. Whenever  $Rsp(p)$  is enabled,  $Do(p)$  can never become enabled unless a  $Rsp(p)$  step eventually occurs.

These properties are satisfied because a request to  $p$  is issued by a  $Req(p)$  step, executed by a  $Do(p)$  step, and responded to by a  $Rsp(p)$  step; and then, the next request to  $p$  can be issued by a  $Req(p)$  step. Each of these steps becomes possible (the action enabled) only after the preceding one occurs.

Let's now show that DR1 and DR2 imply that the conjunction of weak fairness of  $Do(p)$  and of  $Rsp(p)$  is equivalent to weak fairness of  $Do(p) \vee Rsp(p)$ . For compactness, and to emphasize the generality of what we're doing, let's replace  $Do(p)$ ,  $Rsp(p)$ , and  $vars$  by  $A$ ,  $B$ , and  $v$ , respectively.

First, we must restate DR1 and DR2 as temporal formulas. The basic form of DR1 and DR2 is

Whenever  $F$  is true,  $G$  can never be true unless  $H$  is eventually true.

This is expressed in temporal logic as  $\square(F \Rightarrow \square \neg G \vee \diamond H)$ . (The assertion “ $P$  unless  $Q$ ” just means  $P \vee Q$ .) Adding suitable subscripts, we can therefore write DR1 and DR2 in temporal logic as

$$\begin{aligned} DR1 &\triangleq \square(\text{ENABLED } \langle A \rangle_v \Rightarrow \square \neg \text{ENABLED } \langle B \rangle_v \vee \diamond \langle A \rangle_v) \\ DR2 &\triangleq \square(\text{ENABLED } \langle B \rangle_v \Rightarrow \square \neg \text{ENABLED } \langle A \rangle_v \vee \diamond \langle B \rangle_v) \end{aligned}$$

Our goal is to prove

$$(8.20) \quad DR1 \wedge DR2 \Rightarrow (\text{WF}_v(A) \wedge \text{WF}_v(B) \equiv \text{WF}_v(A \vee B))$$

This is complicated, so we split the proof into pieces. As in the proof of (8.11) in Section 8.4 above, we prove an equivalence by proving two implications. To prove (8.20), we prove the two theorems

$$DR1 \wedge DR2 \wedge \text{WF}_v(A) \wedge \text{WF}_v(B) \Rightarrow \text{WF}_v(A \vee B)$$

$$DR1 \wedge DR2 \wedge \text{WF}_v(A \vee B) \Rightarrow \text{WF}_v(A) \wedge \text{WF}_v(B)$$

We prove them by showing that they are true for an arbitrary behavior  $\sigma$ . In other words, we prove

$$(8.21) \quad (\sigma \models DR1 \wedge DR2 \wedge \text{WF}_v(A) \wedge \text{WF}_v(B)) \Rightarrow (\sigma \models \text{WF}_v(A \vee B))$$

$$(8.22) \quad (\sigma \models DR1 \wedge DR2 \wedge \text{WF}_v(A \vee B)) \Rightarrow (\sigma \models \text{WF}_v(A) \wedge \text{WF}_v(B))$$

These formulas seem daunting. Whenever you have trouble proving something, try a proof by contradiction; it gives you an extra hypothesis for free—namely, the negation of what you're trying to prove. Proofs by contradiction are especially useful in temporal logic. To prove (8.21) and (8.22) by contradiction, we need to compute  $\neg(\sigma \models \text{WF}_v(C))$  for an action  $C$ . From the definition (8.7) of  $\text{WF}$ , we easily get

$$(8.23) \quad (\sigma \models \text{WF}_v(C)) \equiv \forall n \in \text{Nat} : (\sigma^{+n} \models \square \text{ENABLED } \langle C \rangle_v) \Rightarrow (\sigma^{+n} \models \diamond \langle C \rangle_v)$$

This and the tautology

$$\neg(\forall x \in S : P \Rightarrow Q) \equiv (\exists x \in S : P \wedge \neg Q)$$

of predicate logic yields

$$(8.24) \quad \neg(\sigma \models \text{WF}_v(C)) \equiv \exists n \in \text{Nat} : (\sigma^{+n} \models \square \text{ENABLED } \langle C \rangle_v) \wedge \neg(\sigma^{+n} \models \diamond \langle C \rangle_v)$$

We also need two further results, both of which are derived from the tautology  $\langle A \vee B \rangle_v \equiv \langle A \rangle_v \vee \langle B \rangle_v$ . Combining this tautology with the temporal tautology  $\diamond(F \vee G) \equiv \diamond F \vee \diamond G$  yields

$$(8.25) \quad \diamond \langle A \vee B \rangle_v \equiv \diamond \langle A \rangle_v \vee \diamond \langle B \rangle_v$$

Combining the tautology with the observation that an action  $C \vee D$  is enabled iff action  $C$  or action  $D$  is enabled yields

$$(8.26) \quad \text{ENABLED } \langle A \vee B \rangle_v \equiv \text{ENABLED } \langle A \rangle_v \vee \text{ENABLED } \langle B \rangle_v$$

We can now prove (8.21) and (8.22). To prove (8.21), we assume that  $\sigma$  satisfies *DR1*, *DR2*,  $\text{WF}_v(A)$ , and  $\text{WF}_v(B)$ , but it does not satisfy  $\text{WF}_v(A \vee B)$ , and we obtain a contradiction. By (8.24), the assumption that  $\sigma$  does not satisfy  $\text{WF}_v(A \vee B)$  means that there exists some number  $n$  such that

$$(8.27) \quad \sigma^{+n} \models \square \text{ENABLED } \langle A \vee B \rangle_v$$

$$(8.28) \quad \neg(\sigma^{+n} \models \diamond \langle A \vee B \rangle_v)$$

We obtain a contradiction from (8.27) and (8.28) as follows:

$$1. \quad \neg(\sigma^{+n} \models \diamond \langle A \rangle_v) \wedge \neg(\sigma^{+n} \models \diamond \langle B \rangle_v)$$

PROOF: By (8.28) and (8.25), using the tautology  $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$ .

$$2. \quad (a) \quad (\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v)$$

$$(b) \quad (\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \square \neg \text{ENABLED } \langle A \rangle_v)$$

PROOF: By definition of *DR1*, the assumption  $\sigma \models \text{DR1}$  implies

$$(\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \Rightarrow$$

$$(\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v) \vee (\sigma^{+n} \models \diamond \langle A \rangle_v)$$

and part (a) then follows from 1. The proof of (b) is similar.

$$3. \quad (a) \quad (\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \square \text{ENABLED } \langle A \rangle_v)$$

$$(b) \quad (\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \square \text{ENABLED } \langle B \rangle_v)$$

PROOF: Part (a) follows from 2(a), (8.27), (8.26), and the temporal tautology

$$\square(F \vee G) \wedge \square \neg G \Rightarrow \square F$$

The proof of part (b) is similar.

4. (a)  $(\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \diamond \langle A \rangle_v)$   
 (b)  $(\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v) \Rightarrow (\sigma^{+n} \models \diamond \langle B \rangle_v)$

PROOF: The assumption  $\sigma \models \text{WF}_v(A)$  and (8.23) imply

$$(\sigma^{+n} \models \square \text{ENABLED } \langle A \rangle_v) \Rightarrow (\sigma^{+n} \models \diamond \langle A \rangle_v)$$

Part (a) follows from this and 3(a). The proof of part (b) is similar.

5.  $(\sigma^{+n} \models \diamond \langle A \rangle_v) \vee (\sigma^{+n} \models \diamond \langle B \rangle_v)$

PROOF: Since  $\square F$  implies  $F$ , for any  $F$ , (8.27) and (8.26) imply

$$(\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v) \vee (\sigma^{+n} \models \text{ENABLED } \langle B \rangle_v)$$

Step 5 then follows by propositional logic from step 4.

Steps 1 and 5 provide the required contradiction.

We can prove (8.22) by assuming that  $\sigma$  satisfies *DR1*, *DR2*, and  $\text{WF}_v(A \vee B)$ , and then proving that it satisfies  $\text{WF}_v(A)$  and  $\text{WF}_v(B)$ . We prove only that it satisfies  $\text{WF}_v(A)$ ; the proof for  $\text{WF}_v(B)$  is similar. The proof is by contradiction; we assume that  $\sigma$  does not satisfy  $\text{WF}_v(A)$  and obtain a contradiction. By (8.24), the assumption that  $\sigma$  does not satisfy  $\text{WF}_v(A)$  means that there exists some number  $n$  such that

$$(8.29) \quad \sigma^{+n} \models \square \text{ENABLED } \langle A \rangle_v$$

$$(8.30) \quad \neg(\sigma^{+n} \models \diamond \langle A \rangle_v)$$

We obtain the contradiction as follows:

1.  $\sigma^{+n} \models \diamond \langle A \vee B \rangle_v$

PROOF: From (8.29) and (8.26) we deduce  $\sigma^{+n} \models \square \text{ENABLED } \langle A \vee B \rangle_v$ . By the assumption  $\sigma \models \text{WF}_v(A \vee B)$  and (8.23), this implies  $\sigma^{+n} \models \diamond \langle A \vee B \rangle_v$ .

2.  $\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v$

PROOF: From (8.29) we deduce  $\sigma^{+n} \models \text{ENABLED } \langle A \rangle_v$ , which by the assumption  $\sigma \models \text{DR1}$  and the definition of *DR1* implies

$$(\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v) \vee (\sigma^{+n} \models \diamond \langle A \rangle_v)$$

The assumption (8.30) then implies  $\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v$ .

3.  $\neg(\sigma^{+n} \models \diamond \langle B \rangle_v)$

PROOF: The definition of *ENABLED* implies  $\neg \text{ENABLED } \langle B \rangle_v \Rightarrow \neg \langle B \rangle_v$ . (A  $\langle B \rangle_v$  step can occur only when it is enabled.) From this, simple temporal reasoning implies

$$(\sigma^{+n} \models \square \neg \text{ENABLED } \langle B \rangle_v) \Rightarrow \neg(\sigma^{+n} \models \diamond \langle B \rangle_v)$$

(A formal proof uses the Implies Generalization Rule and the tautology  $\square \neg F \equiv \neg \diamond F$ .) We then deduce  $\neg(\sigma^{+n} \models \diamond \langle B \rangle_v)$  from 2.

4.  $\neg(\sigma^{+n} \models \diamond \langle A \vee B \rangle_v)$

PROOF: By (8.30), 3, and (8.25), using the tautology  $\neg P \wedge \neg Q \equiv \neg(P \vee Q)$ .

Steps 1 and 4 provide the necessary contradiction. This completes our proof of (8.22), which completes our proof of (8.20).

### 8.5.3 A Generalization

Formula (8.20) provides a rule for replacing the conjunction of weak fairness requirements on two actions with weak fairness of their disjunction. We now generalize it from two actions  $A$  and  $B$  to  $n$  actions  $A_1, \dots, A_n$ . The generalization of  $DR1$  and  $DR2$  is

$$DR(i, j) \triangleq \square(\text{ENABLED } \langle A_i \rangle_v \Rightarrow \square \neg \text{ENABLED } \langle A_j \rangle_v \vee \diamond \langle A_i \rangle_v)$$

If we substitute  $A_1$  for  $A$  and  $A_2$  for  $B$ , then  $DR1$  becomes  $DR(1, 2)$  and  $DR2$  becomes  $DR(2, 1)$ . The generalization of (8.20) is

$$(8.31) (\forall i, j \in 1 \dots n : (i \neq j) \Rightarrow DR(i, j)) \Rightarrow (\text{WF}_v(A_1) \wedge \dots \wedge \text{WF}_v(A_n) \equiv \text{WF}_v(A_1 \vee \dots \vee A_n))$$

To decide if you can replace the conjunction of weak fairness conditions by a single one in a specification, you will probably find it easier to use the following informal statement of (8.31):

**WF Conjunction Rule** If  $A_1, \dots, A_n$  are actions such that, for any distinct  $i$  and  $j$ , whenever  $\langle A_i \rangle_v$  is enabled,  $\langle A_j \rangle_v$  cannot become enabled unless an  $\langle A_i \rangle_v$  step occurs, then  $\text{WF}_v(A_1) \wedge \dots \wedge \text{WF}_v(A_n)$  is equivalent to  $\text{WF}_v(A_1 \vee \dots \vee A_n)$ .

Perhaps the best way to think of this rule is as an assertion about an arbitrary individual behavior  $\sigma$ . Its hypothesis is then that  $\sigma \models DR(i, j)$  holds for all distinct  $i$  and  $j$ ; its conclusion is

$$\sigma \models (\text{WF}_v(A_1) \wedge \dots \wedge \text{WF}_v(A_n) \equiv \text{WF}_v(A_1 \vee \dots \vee A_n))$$

To replace  $\text{WF}_v(A_1) \wedge \dots \wedge \text{WF}_v(A_n)$  by  $\text{WF}_v(A_1 \vee \dots \vee A_n)$  in a specification, you have to check that any behavior satisfying the safety part of the specification also satisfies  $DR(i, j)$ , for all distinct  $i$  and  $j$ .

Conjunction and disjunction are special cases of quantification:

$$F_1 \vee \dots \vee F_n \equiv \exists i \in 1 \dots n : F_i$$

$$F_1 \wedge \dots \wedge F_n \equiv \forall i \in 1 \dots n : F_i$$

We can therefore easily restate the WF Conjunction Rule as a condition on when  $\forall i \in S : \text{WF}_v(A_i)$  and  $\text{WF}_v(\exists i \in S : A_i)$  are equivalent, for a finite set  $S$ . The resulting rule is actually valid for any set  $S$ :

**WF Quantifier Rule** If, for all  $i \in S$ , the  $A_i$  are actions such that, for any distinct  $i$  and  $j$  in  $S$ , whenever  $\langle A_i \rangle_v$  is enabled,  $\langle A_j \rangle_v$  cannot become enabled unless an  $\langle A_i \rangle_v$  step occurs, then  $\forall i \in S : \text{WF}_v(A_i)$  is equivalent to  $\text{WF}_v(\exists i \in S : A_i)$ .

## 8.6 Strong Fairness

We define  $SF_v(A)$ , *strong fairness* of action  $A$ , to be either of the following two equivalent formulas:

$$(8.32) \quad \diamond \square (\neg \text{ENABLED } \langle A \rangle_v) \vee \square \diamond \langle A \rangle_v$$

$$(8.33) \quad \square \diamond \text{ENABLED } \langle A \rangle_v \Rightarrow \square \diamond \langle A \rangle_v$$

Intuitively, these two formulas assert

(8.32)  $A$  is eventually disabled forever, or infinitely many  $A$  steps occur.

(8.33) If  $A$  is infinitely often enabled, then infinitely many  $A$  steps occur.

The proof that (8.32) and (8.33) are equivalent is similar to the proof on page 98 that the two formulations (8.8) and (8.9) of  $WF_v(A)$  are equivalent.

Definition (8.32) of  $SF_v(A)$  is obtained from definition (8.8) of  $WF_v(A)$  by replacing  $\square \diamond (\neg \text{ENABLED } \langle A \rangle_v)$  with  $\diamond \square (\neg \text{ENABLED } \langle A \rangle_v)$ . Since  $\diamond \square F$  (eventually always  $F$ ) is stronger than (implies)  $\square \diamond F$  (infinitely often  $F$ ) for any formula  $F$ , strong fairness is stronger than weak fairness. We can express weak and strong fairness as follows:

- Weak fairness of  $A$  asserts that an  $A$  step must eventually occur if  $A$  is *continuously* enabled.
- Strong fairness of  $A$  asserts that an  $A$  step must eventually occur if  $A$  is *continually* enabled.

*Continuously* means without interruption. *Continually* means repeatedly, possibly with interruptions.

Strong fairness need not be strictly stronger than weak fairness. Weak and strong fairness of an action  $A$  are equivalent iff  $A$  infinitely often disabled implies that either  $A$  eventually becomes forever disabled, or else infinitely many  $A$  steps occur. This is expressed formally by the tautology

$$\begin{aligned} (\text{WF}_v(A) \equiv \text{SF}_v(A)) &\equiv \\ (\square \diamond (\neg \text{ENABLED } \langle A \rangle_v) \Rightarrow \diamond \square (\neg \text{ENABLED } \langle A \rangle_v) \vee \square \diamond \langle A \rangle_v) & \end{aligned}$$

In the channel example, weak and strong fairness of  $Rcv$  are equivalent because  $Spec$  implies that, once enabled,  $Rcv$  can be disabled only by a  $Rcv$  step. Hence, if  $Rcv$  is disabled infinitely often, then it either eventually remains disabled forever, or else it is disabled infinitely often by  $Rcv$  steps.

The analogs of the WF Conjunction and WF Quantifier Rules (page 105) hold for strong fairness—for example:

**SF Conjunction Rule** If  $A_1, \dots, A_n$  are actions such that, for any distinct  $i$  and  $j$ , whenever action  $A_i$  is enabled, action  $A_j$  cannot become enabled until an  $A_i$  step occurs, then  $\text{SF}_v(A_1) \wedge \dots \wedge \text{SF}_v(A_n)$  is equivalent to  $\text{SF}_v(A_1 \vee \dots \vee A_n)$ .

Strong fairness can be more difficult to implement than weak fairness, and it is a less common requirement. A strong fairness condition should be used in a specification only if it is needed. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness.

Liveness properties can be subtle. Expressing them with *ad hoc* temporal formulas can lead to errors. We will specify liveness as the conjunction of weak and/or strong fairness properties whenever possible—and it almost always is possible. Having a uniform way of expressing liveness makes specifications easier to understand. Section 8.9.2 below discusses an even more compelling reason for using fairness to specify liveness.

## 8.7 The Write-Through Cache

Let's now add liveness to the write-through cache, specified in Figure 5.5 on pages 57–59. We want our specification to guarantee that every request eventually receives a response, without requiring that any requests are issued. This requires fairness on all the actions that make up the next-state action *Next* except for the following:

- A  $Req(p)$  action, which issues a request.
- An  $Evict(p, a)$  action, which evicts an address from the cache.
- A  $MemQWr$  action, if  $memQ$  contains only write requests and is not full (has fewer than  $QLen$  elements). Since a response to a write request can be issued before the value is written to memory, failing to execute a  $MemQWr$  action can prevent a response only if it prevents the dequeuing of a read operation in  $memQ$  or the enqueueing of an operation (because  $memQ$  is full).

For simplicity, let's require fairness for the  $MemQWr$  action too; we'll weaken this requirement later. Our liveness condition then has to assert fairness of the actions

$MemQWr \quad MemQRd \quad Rsp(p) \quad RdMiss(p) \quad DoRd(p) \quad DoWr(p)$

for all  $p$  in  $Proc$ . We now must decide whether to assert weak or strong fairness for these actions. Weak and strong fairness are equivalent for an action that, once enabled, remains enabled until it is executed. This is the case for all of these actions except  $DoRd(p)$ ,  $RdMiss(p)$ , and  $DoWr(p)$ .

The  $DoRd(p)$  action can be disabled by an  $Evict$  step that evicts the requested data from the cache. In this case, fairness of other actions should imply that the data will eventually be returned to the cache, re-enabling  $DoRd(p)$ .

The data cannot be evicted again until the  $DoRd(p)$  action is executed, and weak fairness then suffices to ensure that the necessary  $DoRd(p)$  step eventually occurs.

The  $RdMiss(p)$  and  $DoWr(p)$  actions append a request to the  $memQ$  queue. They are disabled if that queue is full. A  $RdMiss(p)$  or  $DoWr(p)$  could be enabled and then become disabled because a  $RdMiss(q)$  or  $DoWr(q)$ , for a different processor  $q$ , appends a request to  $memQ$ . We therefore need strong fairness for the  $RdMiss(p)$  and  $DoWr(p)$  actions. So, the fairness conditions we need are

Weak Fairness for  $Rsp(p)$ ,  $DoRd(p)$ ,  $MemQWr$ , and  $MemQRd$

Strong Fairness for  $RdMiss(p)$  and  $DoWr(p)$ .

As before, let's define  $vars$  to be the tuple of all variables.

$$vars \triangleq (memInt, wmem, buf, ctl, cache, memQ)$$

We could just write the liveness condition as

$$(8.34) \wedge \forall p \in Proc : \wedge WF_{vars}(Rsp(p)) \wedge WF_{vars}(DoRd(p)) \\ \wedge SF_{vars}(RdMiss(p)) \wedge SF_{vars}(DoWr(p)) \\ \wedge WF_{vars}(MemQWr) \wedge WF_{vars}(MemQRd)$$

However, I prefer replacing the conjunction of fairness conditions by a single fairness condition on a disjunction, as we did in Section 8.5 for the memory specification. The WF and SF Conjunction Rules (pages 105 and 106) imply that the liveness condition (8.34) can be rewritten as

$$(8.35) \wedge \forall p \in Proc : \wedge WF_{vars}(Rsp(p) \vee DoRd(p)) \\ \wedge SF_{vars}(RdMiss(p) \vee DoWr(p)) \\ \wedge WF_{vars}(MemQWr \vee MemQRd)$$

We can now try to simplify (8.35) by moving the quantifier inside the WF and SF formulas. First, because  $\forall$  distributes over  $\wedge$ , we can rewrite the first conjunct of (8.35) as

$$(8.36) \wedge \forall p \in Proc : WF_{vars}(Rsp(p) \vee DoRd(p)) \\ \wedge \forall p \in Proc : SF_{vars}(RdMiss(p) \vee DoWr(p))$$

We can now try to apply the WF Quantifier Rule (page 105) to the first conjunct of (8.36) and the corresponding SF Quantifier Rule to its second conjunct. However, the WF quantifier rule doesn't apply to the first conjunct. It's possible for both  $Rsp(p) \vee DoRd(p)$  and  $Rsp(q) \vee DoRd(q)$  to be enabled at the same time, for two different processors  $p$  and  $q$ . The formula

$$(8.37) \text{WF}_{\text{vars}}(\exists p \in \text{Proc} : \text{Rsp}(p) \vee \text{DoRd}(p))$$

is satisfied by any behavior in which infinitely many  $\text{Rsp}(p)$  and  $\text{DoRd}(p)$  actions occur for some processor  $p$ . In such a behavior,  $\text{Rsp}(q)$  could be enabled for some other processor  $q$  without an  $\text{Rsp}(q)$  step ever occurring, making  $\text{WF}_{\text{vars}}(\text{Rsp}(q) \vee \text{DoRd}(q))$  false, which implies that the first conjunct of (8.36) is false. Hence, (8.37) is not equivalent to the first conjunct of (8.36). Similarly, the analogous rule for strong fairness cannot be applied to the second conjunct of (8.36). Formula (8.35) is as simple as we can make it.

Let's return to the observation that we don't have to execute  $\text{MemQWr}$  if the  $\text{memQ}$  queue contains only write requests and is not full. In other words, we have to execute  $\text{MemQWr}$  only if  $\text{memQ}$  is full or contains a read request. Let's define

$$\begin{aligned} QCond &\triangleq \vee \text{Len}(\text{memQ}) = QLen \\ &\vee \exists i \in 1 \dots \text{Len}(\text{memQ}) : \text{memQ}[i][2].op = \text{"Rd"} \end{aligned}$$

so we need eventually execute a  $\text{MemQWr}$  action only when it's enabled and  $QCond$  is true, which is the case iff the action  $QCond \wedge \text{MemQWr}$  is enabled. In this case, a  $\text{MemQWr}$  step is a  $QCond \wedge \text{MemQWr}$  step. Hence, it suffices to require weak fairness of the action  $QCond \wedge \text{MemQWr}$ . We can therefore replace the second conjunct of (8.35) with

$$\text{WF}_{\text{vars}}((QCond \wedge \text{MemQWr}) \vee \text{MemQRd})$$

We would do this if we wanted the specification to describe the weakest liveness condition that implements the memory specification's liveness condition. However, if the specification were a description of an actual device, then that device would probably implement weak fairness on all  $\text{MemQWr}$  actions, so we would take (8.35) as the liveness condition.

## 8.8 Quantification

Section 8.1 describes the meaning of ordinary quantification of temporal formulas. For example, the meaning of the formula  $\forall r : F$ , for any temporal formula  $F$ , is defined by

$$\sigma \models (\forall r : F) \triangleq \forall r : (\sigma \models F)$$

where  $\sigma$  is any behavior.

The symbol  $r$  in  $\exists r : F$  is usually called a bound variable. But we've been using the term *variable* to mean something else—something that's declared by a VARIABLE statement in a module. The bound “variable”  $r$  is actually a constant

in these formulas—a value that is the same in every state of the behavior.<sup>5</sup> For example, the formula  $\exists r : \square(x = r)$  asserts that  $x$  has the same value in every state of a behavior.

Bounded quantification over a constant set  $S$  is defined by

$$\sigma \models (\forall r \in S : F) \triangleq (\forall r \in S : \sigma \models F)$$

$$\sigma \models (\exists r \in S : F) \triangleq (\exists r \in S : \sigma \models F)$$

The symbol  $r$  is declared to be a constant in formula  $F$ . The expression  $S$  lies outside the scope of the declaration of  $r$ , so the symbol  $r$  cannot occur in  $S$ . It's easy to define the meanings of these formulas even if  $S$  is not a constant—for example, by letting  $\exists r \in S : F$  equal  $\exists r : (r \in S) \wedge F$ . However, for nonconstant  $S$ , it's better to write  $\exists r : (r \in S) \wedge F$  explicitly.

It's also easy to define the meaning of CHOOSE as a temporal operator. We can just let  $\sigma \models (\text{CHOOSE } r : F)$  be an arbitrary constant value  $r$  such that  $\sigma \models F$  equals TRUE, if such an  $r$  exists. However, a temporal CHOOSE operator is not needed for writing specifications, so CHOOSE  $r : F$  is not a legal TLA<sup>+</sup> formula if  $F$  is a temporal formula.

We now come to the temporal existential quantifier  $\exists$ . In the formula  $\exists x : F$ , the symbol  $x$  is declared to be a variable in  $F$ . Unlike  $\exists r : F$ , which asserts the existence of a single value  $r$ , the formula  $\exists x : F$  asserts the existence of a value for  $x$  in each state of a behavior. For example, if  $y$  is a variable, then the formula  $\exists x : \square(x \in y)$  asserts that  $y$  always has some element  $x$ , so  $y$  is always a nonempty set. However, the element  $x$  could be different in different states, so the values of  $y$  in different states could be disjoint.

We have been using  $\exists$  as a hiding operator, thinking of  $\exists x : F$  as  $F$  with variable  $x$  hidden. The precise definition of  $\exists$  is a bit tricky because, as discussed in Section 8.1, the formula  $\exists x : F$  should be invariant under stuttering. Intuitively,  $\exists x : F$  is satisfied by a behavior  $\sigma$  iff  $F$  is satisfied by a behavior  $\tau$  that is obtained from  $\sigma$  by adding and/or deleting stuttering steps and changing the value of  $x$ . A precise definition appears in Section 16.2.4 (page 314). However, for writing specifications, you can simply think of  $\exists x : F$  as  $F$  with  $x$  hidden.

TLA also has a temporal universal quantifier  $\forall$ , defined by

$$\forall x : F \triangleq \neg \exists x : \neg F$$

This operator is hardly ever used. TLA<sup>+</sup> does not allow bounded versions of the operators  $\exists$  and  $\forall$ .

---

<sup>5</sup>Logicians use the term *flexible variable* for a TLA variable, and the term *rigid variable* for a symbol like  $r$  that represents a constant.

## 8.9 Temporal Logic Examined

### 8.9.1 A Review

Let's look at the shapes of the specifications that we've written so far. We started with the simple form

$$(8.38) \quad \text{Init} \wedge \square[\text{Next}]_{\text{vars}}$$

where *Init* is the initial predicate, *Next* the next-state action, and *vars* the tuple of all variables. This kind of specification is, in principle, quite straightforward. We then introduced hiding, using  $\exists$  to bind variables that should not appear in the specification. Those bound variables, also called *hidden* or *internal* variables, serve only to help describe how the values of the free variables (also called visible variables) change.

Hiding variables is easy enough, and it is mathematically elegant and philosophically satisfying. However, in practice, it doesn't make much difference to a specification. A comment can also tell a reader that a variable should be regarded as internal. Explicit hiding allows implementation to mean implication. A lower-level specification that describes an implementation can be expected to imply a higher-level specification only if the higher-level specification's internal variables, whose values don't really matter, are explicitly hidden. Otherwise, implementation means implementation under a refinement mapping. (See Section 5.8.) However, as explained in Section 10.8 below, implementation often involves a refinement of the visible variables as well.

To express liveness, the specification (8.38) is strengthened to the form

$$(8.39) \quad \text{Init} \wedge \square[\text{Next}]_{\text{vars}} \wedge \text{Liveness}$$

where *Liveness* is the conjunction of formulas of the form  $\text{WF}_{\text{vars}}(A)$  and/or  $\text{SF}_{\text{vars}}(A)$ , for actions *A*. (I'm considering universal quantification to be a form of conjunction.)

### 8.9.2 Machine Closure

In the specifications of the form (8.39) we've written so far, the actions *A* whose fairness properties appear in formula *Liveness* have one thing in common: they are all *subactions* of the next-state action *Next*. An action *A* is a subaction of *Next* iff every *A* step is a *Next* step. Equivalently, *A* is a subaction of *Next* iff *A* implies *Next*.<sup>6</sup> In almost all specifications of the form (8.39), formula *Liveness*

---

<sup>6</sup>We can also use the following weaker definition of subaction: *A* is a subaction of formula (8.38) iff, for every state *s* of every behavior satisfying (8.38), if *A* is enabled in state *s* then *Next*  $\wedge$  *A* is also enabled in *s*.

should be the conjunction of weak and/or strong fairness formulas for subactions of *Next*. I'll now explain why.

When we look at the specification (8.39), we expect *Init* to constrain the initial state, *Next* to constrain what steps may occur, and *Liveness* to describe only what must eventually happen. However, consider the following formula:

$$(8.40) \quad (x = 0) \wedge \square[x' = x + 1]_x \wedge \text{WF}_x((x > 99) \wedge (x' = x - 1))$$

The first two conjuncts of (8.40) assert that  $x$  is initially 0 and that any step either increments  $x$  by 1 or leaves it unchanged. Hence, they imply that if  $x$  ever exceeds 99, then it forever remains greater than 99. The weak fairness property asserts that, if this happens, then  $x$  must eventually be decremented by 1—contradicting the second conjunct. Hence, (8.40) implies that  $x$  can never exceed 99, so it is equivalent to

$$(x = 0) \wedge \square[(x < 99) \wedge (x' = x + 1)]_x$$

Conjoining the weak fairness property to the first two conjuncts of (8.40) forbids an  $x' = x + 1$  step when  $x = 99$ .

A specification of the form (8.39) is called *machine closed* iff the conjunct *Liveness* constrains neither the initial state nor what steps may occur. A more general way to express this is as follows. Let a finite behavior be a finite sequence of states.<sup>7</sup> We say that a finite behavior  $\sigma$  satisfies a safety property  $S$  iff the behavior obtained by adding infinitely many stuttering steps to the end of  $\sigma$  satisfies  $S$ . If  $S$  is a safety property, then we define the pair of formulas  $S, L$  to be machine closed iff every finite behavior that satisfies  $S$  can be extended to an infinite behavior that satisfies  $S \wedge L$ . We call (8.39) machine closed if the pair of formulas  $\text{Init} \wedge \square[\text{Next}]_{\text{vars}}$ , *Liveness* is machine closed.

We seldom want to write a specification that isn't machine closed. If we do write one, it's usually by mistake. Specification (8.39) is guaranteed to be machine closed if *Liveness* is the conjunction of weak and/or strong fairness properties for subactions of *Next*.<sup>8</sup> This condition doesn't hold for specification (8.40), which is not machine closed, because  $(x > 99) \wedge (x' = x - 1)$  is not a subaction of  $x' = x + 1$ .

Liveness requirements are philosophically satisfying. A specification of the form (8.38), which specifies only a safety property, allows behaviors in which the system does nothing. Therefore, the specification is satisfied by a system that does nothing. Expressing liveness requirements with fairness properties is less satisfying. These properties are subtle and it's easy to get them wrong.

---

<sup>7</sup>A finite behavior therefore isn't a behavior, which is an infinite sequence of states. Mathematicians often abuse language in this way.

<sup>8</sup>More precisely, this is the case for a finite or countably infinite conjunction of properties of the form  $\text{WF}_v(A)$  and/or  $\text{SF}_v(A)$ , where each  $\langle A \rangle_v$  is a subaction of *Next*. This result also holds for the weaker definition of subaction in the footnote on the preceding page.

It requires some thought to determine that the liveness condition for the write-through cache, formula (8.35) on page 108, does imply that every request receives a reply.

It's tempting to express liveness properties more directly, without using fairness properties. For example, it's easy to write a temporal formula asserting for the write-through cache that every request receives a response. When processor  $p$  issues a request, it sets  $ctl[p]$  to "rdy". We just have to assert that, for every processor  $p$ , whenever a state in which  $ctl[p] = \text{"rdy"}$  is true occurs, there will eventually be a  $Rsp(p)$  step:

$$(8.41) \forall p \in Proc : \square((ctl[p] = \text{"rdy"}) \Rightarrow \diamond \langle Rsp(p) \rangle_{vars})$$

While such formulas are appealing, they are dangerous. It's very easy to make a mistake and write a specification that isn't machine closed.

Except in unusual circumstances, you should express liveness with fairness properties for subactions of the next-state action. These are the most straightforward specifications, and hence the easiest to write and to understand. Most system specifications, even if very detailed and complicated, can be written in this straightforward manner. The exceptions are usually in the realm of subtle, high-level specifications that attempt to be very general. An example of such a specification appears in Section 11.2.

### 8.9.3 Machine Closure and Possibility

Machine closure can be thought of as a possibility condition. For example, machine closure of the pair  $S, \square \diamond \langle A \rangle_v$  asserts that for every finite behavior  $\sigma$  satisfying  $S$ , it is possible to extend  $\sigma$  to an infinite behavior satisfying  $S$  in which infinitely many  $\langle A \rangle_v$  actions occur. If we regard  $S$  as a system specification, so a behavior that satisfies  $S$  represents a possible execution of the system, then we can restate machine closure of  $S, \square \diamond \langle A \rangle_v$  as follows: in any system execution, it is always possible for infinitely many  $\langle A \rangle_v$  actions to occur.

TLA specifications express safety and liveness properties, not possibility properties. A safety property asserts that something is impossible—for example, the system cannot take a step that doesn't satisfy the next-state action. A liveness property asserts that something must eventually happen. System requirements are sometimes stated informally in terms of what is possible. Most of the time, when examined rigorously, these requirements can be expressed with liveness and/or safety properties. (The most notable exceptions are statistical properties, such as assertions about the probability that something happens.) We are never interested in specifying that something *might* happen. It's never useful to know that the system *might* produce the right answer. We never have to specify that the user *might* type an "a"; we must specify what happens if he does.

Machine closure is a property of a pair of formulas, not of a system. Although a possibility property is never a useful assertion about a system, it can be a useful assertion about a specification. A specification  $S$  of a system with keyboard input should always allow the user to type an “a”. So, every finite behavior satisfying  $S$  should be extendable to an infinite behavior satisfying  $S$  in which infinitely many “a”s are typed. If the action  $\langle A \rangle_v$  represents the typing of an “a”, then saying that the user should always be able to type infinitely many “a”s is equivalent to saying that the pair  $S, \square \diamond \langle A \rangle_v$  should be machine closed. If  $S, \square \diamond \langle A \rangle_v$  isn’t machine closed, then it could become impossible for the user ever to type an “a”. Unless the system is allowed to lock the keyboard, this would mean that there was something wrong with the specification.

This kind of possibility property can be proved. For example, to prove that it’s always possible for the user to type infinitely many “a”s, we show that conjoining suitable fairness conditions on the input actions implies that the user *must* type infinitely many “a”s. However, proofs of this kind of simple property don’t seem to be worth the effort. When writing a specification, you should make sure that possibilities allowed by the real system are allowed by the specification. Once you are aware of what should be possible, you will usually have little trouble ensuring that the specification makes it possible. You should also make sure that what the system *must* do is implied by the specification’s fairness conditions. This can be more difficult.

### 8.9.4 Refinement Mappings and Fairness

Section 5.8 (page 62) describes how to prove that the write-through memory implements the memory specification. We have to prove  $Spec \Rightarrow \overline{ISpec}$ , where  $Spec$  is the specification of the write-through memory,  $ISpec$  is the internal specification of the memory (with the internal variables made visible), and, for any formula  $F$ , we let  $\overline{F}$  mean  $F$  with expressions  $omem$ ,  $octl$ , and  $obuf$  substituted for the variables  $mem$ ,  $ctl$ , and  $buf$ . We could rewrite this implication as (5.3) because substitution (overbarring) distributes over operators like  $\wedge$  and  $\square$ , so we had

$$\begin{aligned}
 & \overline{Init} \wedge \square[\overline{INext}]_{\langle memInt, mem, ctl, buf \rangle} \\
 & \equiv \overline{Init} \wedge \square[\overline{INext}]_{\langle memInt, mem, ctl, buf \rangle} && \text{Because } \overline{\quad} \text{ distributes over } \wedge. \\
 & \equiv \overline{Init} \wedge \square[\overline{INext}]_{\langle \overline{memInt}, \overline{mem}, \overline{ctl}, \overline{buf} \rangle} && \text{Because } \overline{\quad} \text{ distributes over } \square[\dots]. \\
 & \equiv \overline{Init} \wedge \square[\overline{INext}]_{\langle \overline{memInt}, \overline{mem}, \overline{ctl}, \overline{buf} \rangle} && \text{Because } \overline{\quad} \text{ distributes over } \langle \dots \rangle. \\
 & \equiv \overline{Init} \wedge \square[\overline{INext}]_{\langle memInt, \overline{mem}, \overline{ctl}, \overline{buf} \rangle} && \text{Because } \overline{memInt} = memInt.
 \end{aligned}$$

Adding liveness to the specifications adds conjuncts to the formulas  $Spec$  and  $ISpec$ . Suppose we take formula  $Liveness2$ , defined in (8.19) on page 101, as

the liveness property of  $ISpec$ . Then  $\overline{ISpec}$  has the additional term  $\overline{Liveness2}$ , which can be simplified as follows:

$$\begin{aligned} \overline{Liveness2} &\equiv \overline{\forall p \in Proc : WF_{vars}(Do(p) \vee Rsp(p))} && \text{By definition of } \overline{Liveness2}. \\ &\equiv \forall p \in Proc : \overline{WF_{vars}(Do(p) \vee Rsp(p))} && \text{Because } \neg \text{ distributes over } \forall. \end{aligned}$$

But we cannot automatically move the  $\neg$  inside the WF because substitution does not, in general, distribute over ENABLED, and hence it does not distribute over WF or SF. For the specifications and refinement mappings that occur in practice, including this one, simply replacing each  $\overline{WF_v(A)}$  by  $WF_{\overline{v}}(\overline{A})$  and each  $\overline{SF_v(A)}$  by  $SF_{\overline{v}}(\overline{A})$  does give the right result. However, you don't have to depend on this. You can instead expand the definitions of WF and SF to get, for example:

$$\begin{aligned} \overline{WF_v(A)} &\equiv \overline{\square \diamond \neg \text{ENABLED } \langle A \rangle_v \vee \square \diamond \langle A \rangle_v} && \text{By definition of WF.} \\ &\equiv \square \diamond \neg \overline{\text{ENABLED } \langle A \rangle_v} \vee \square \diamond \overline{\langle A \rangle_v} && \text{By distributivity of } \neg. \end{aligned}$$

You can compute the ENABLED predicates “by hand” and then perform the substitution. When computing ENABLED predicates, it suffices to consider only states satisfying the safety part of the specification, which usually means that  $\text{ENABLED } \langle A \rangle_v$  equals  $\text{ENABLED } A$ . You can then compute ENABLED predicates using the following rules:

1.  $\text{ENABLED } (A \vee B) \equiv (\text{ENABLED } A) \vee (\text{ENABLED } B)$ , for any actions  $A$  and  $B$ .
2.  $\text{ENABLED } (P \wedge A) \equiv P \wedge (\text{ENABLED } A)$ , for any state predicate  $P$  and action  $A$ .
3.  $\text{ENABLED } (A \wedge B) \equiv (\text{ENABLED } A) \wedge (\text{ENABLED } B)$ , if  $A$  and  $B$  are actions such that the same variable does not appear primed in both  $A$  and  $B$ .
4.  $\text{ENABLED } (x' = exp) \equiv \text{TRUE}$  and  $\text{ENABLED } (x' \in exp) \equiv (exp \neq \{\})$ , for any variable  $x$  and state function  $exp$ .

For example:

$$\begin{aligned} \overline{\text{ENABLED } (Do(p) \vee Rsp(p))} &\equiv \overline{(ctl[p] = \text{"rdy"}) \vee (ctl[p] = \text{"done"})} && \text{By rules 1-4.} \\ &\equiv (octl[p] = \text{"rdy"}) \vee (octl[p] = \text{"done"}) && \text{By the meaning of } \neg. \end{aligned}$$

### 8.9.5 The Unimportance of Liveness

While philosophically important, in practice the liveness property of (8.39) is not as important as the safety part,  $Init \wedge \square[Next]_{vars}$ . The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when looking for errors, most of your effort should be devoted to examining the safety part.

### 8.9.6 Temporal Logic Considered Confusing

The most general type of specification I've discussed so far has the form

$$(8.42) \exists v_1, \dots, v_n : Init \wedge \square[Next]_{vars} \wedge Liveness$$

where *Liveness* is the conjunction of fairness properties of subactions of *Next*. This is a very restricted class of temporal-logic formulas. Temporal logic is quite expressive, and one can combine its operators in all sorts of ways to express a wide variety of properties. This suggests the following approach to writing a specification: express each property that the system must satisfy with a temporal formula, and then conjoin all these formulas. For example, formula (8.41) above expresses the property of the write-through cache that every request eventually receives a response.

This approach is philosophically appealing. It has just one problem: it's practical for only the very simplest of specifications—and even for them, it seldom works well. The unbridled use of temporal logic produces formulas that are hard to understand. Conjoining several of these formulas produces a specification that is impossible to understand.

The basic form of a TLA specification is (8.42). Most specifications should have this form. We can also use this kind of specification as a building block. Chapters 9 and 10 describe situations in which we write a specification as a conjunction of such formulas. Section 10.7 introduces an additional temporal operator  $\doteqdot$  and explains why we might want to write a specification  $F \doteqdot G$ , where  $F$  and  $G$  have the form (8.42). But such specifications are of limited practical use. Most engineers need only know how to write specifications of the form (8.42). Indeed, they can get along quite well with specifications of the form (8.38) that express only safety properties and don't hide any variables.

# Chapter 9

## Real Time

With a liveness property, we can specify that a system must eventually respond to a request. We cannot specify that it must respond within the next 100 years. To specify timely response, we must use a real-time property.

A system that does not respond within our lifetime isn't very useful, so we might expect real-time specifications to be common. They aren't. Formal specifications are most often used to describe what a system does rather than how long it takes to do it. However, you may someday want to specify real-time properties of a system. This chapter tells you how.

### 9.1 The Hour Clock Revisited

Let's return to our specification of the simple hour clock in Chapter 2, which asserts that the variable *hr* cycles through the values 1 through 12. We now add the requirement that the clock keep correct time. For centuries, scientists have represented the real-time behavior of a system by introducing a variable, traditionally *t*, whose value is a real number that represents time. A state in which  $t = -17.51$  represents a state of the system at time  $-17.51$ , perhaps measured in seconds elapsed since 00:00 UT on 1 January 2000. In TLA<sup>+</sup> specifications, I prefer to use the variable *now* rather than *t*. For linguistic convenience, I will usually assume that the unit of time is the second, though we could just as well choose any other unit.

Unlike sciences such as physics and chemistry, computer science studies systems whose behavior can be described by a sequence of discrete states, rather than by states that vary continuously with time. We consider the hour clock's display to change directly from reading 12 to reading 1, and ignore the continuum of intermediate states that occur in the physical display. This means that we pretend that the change is instantaneous (happens in 0 seconds). So, a

Remember that a state is an assignment of values to all variables.

real-time specification of the clock might allow the step

$$\begin{bmatrix} hr = 12 \\ now = \sqrt{2.47} \end{bmatrix} \rightarrow \begin{bmatrix} hr = 1 \\ now = \sqrt{2.47} \end{bmatrix}$$

The value of *now* advances between changes to *hr*. If we wanted to specify how long it takes the display to change from 12 to 1, we would have to introduce an intermediate state that represents a changing display—perhaps by letting *hr* assume some intermediate value such as 12.5, or by adding a Boolean-valued variable *chg* whose value indicates whether the display is changing. We won’t do this, but will be content to specify an hour clock in which we consider the display to change instantaneously.

The value of *now* changes between changes to *hr*. Just as we represent a continuously varying clock display by a variable whose value changes in discrete steps, we let the value of *now* change in discrete steps. A behavior in which *now* increases in femtosecond increments would be an accurate enough description of continuously changing time for our specification of the hour clock. In fact, there’s no need to choose any particular granularity of time; we can let *now* advance by arbitrary amounts between clock ticks. (Since the value of *hr* is unchanged by steps that change *now*, the requirement that the clock keep correct time will rule out behaviors in which *now* changes by too much in a single step.)

What real-time condition should our hour clock satisfy? We might require that it always display the time correctly to within  $\rho$  seconds, for some real number  $\rho$ . However, this is not typical of the real-time requirements that arise in actual systems. Instead, we require that the clock tick approximately once per hour. More precisely, we require that the interval between ticks be one hour plus or minus  $\rho$  seconds, for some positive number  $\rho$ . Of course, this requirement allows the time displayed by the clock eventually to drift away from the actual time. But that’s what real clocks do if they are not reset.

We could start our specification of the real-time clock from scratch. However, we still want the hour clock to satisfy the specification *HC* of module *HourClock* (Figure 2.1 on page 20). We just want to add an additional real-time requirement. So, we will write the specification as the conjunction of *HC* and a formula requiring that the clock tick every hour, plus or minus  $\rho$  seconds. This requirement is the conjunction of two separate conditions: that the clock tick at most once every  $3600 - \rho$  seconds, and at least once every  $3600 + \rho$  seconds.

To specify these requirements, we introduce a variable that records how much time has elapsed since the last clock tick. Let’s call it *t* for *timer*. The value of *t* is set to 0 by a step that represents a clock tick—namely, by an *HCnxt* step. Any step that represents the passing of *s* seconds should advance *t* by *s*. A step represents the passing of time iff it changes *now*, and such a step represents the passage of  $now' - now$  seconds. So, the change to the timer *t* is described by the action

$$T_{\text{Next}} \triangleq t' = \text{IF } HC_{\text{nxt}} \text{ THEN } 0 \text{ ELSE } t + (now' - now)$$

We let  $t$  initially equal 0, so we consider the initial state to be one in which the clock has just ticked. The specification of how  $t$  changes is then a formula asserting that  $t$  initially equals 0, and that every step is a  $TNext$  step or else leaves unchanged all relevant variables—namely,  $t$ ,  $hr$ , and  $now$ . This formula is

$$Timer \triangleq (t = 0) \wedge \square[TNext]_{\langle t, hr, now \rangle}$$

The requirement that the clock tick at least once every  $3600 + \rho$  seconds means that it's always the case that at most  $3600 + \rho$  seconds have elapsed since the last  $HCnxt$  step. Since  $t$  always equals the elapsed time since the last  $HCnxt$  step, this requirement is expressed by the formula

$$MaxTime \triangleq \square(t \leq 3600 + \rho)$$

(Since we can't measure time with perfect accuracy, it doesn't matter whether we use  $<$  or  $\leq$  in this formula. When we generalize from this example, it is a bit more convenient to use  $\leq$ .)

The requirement that the clock tick at most once every  $3600 - \rho$  seconds means that, whenever an  $HCnxt$  step occurs, at least  $3600 - \rho$  seconds have elapsed since the previous  $HCnxt$  step. This suggests the condition

$$(9.1) \quad \square(HCnxt \Rightarrow (t \geq 3600 - \rho))$$

In the generalization,  $\geq$  will be more convenient than  $>$ .

However, (9.1) isn't a legal TLA formula because  $HCnxt \Rightarrow \dots$  is an action (a formula containing primes), and a TLA formula asserting that an action is always true must have the form  $\square[A]_v$ . We don't care about steps that leave  $hr$  unchanged, so we can replace (9.1) by the TLA formula

$$MinTime \triangleq \square[HCnxt \Rightarrow (t \geq 3600 - \rho)]_{hr}$$

The desired real-time constraint on the clock is expressed by the conjunction of these three formulas:

$$HCTime \triangleq Timer \wedge MaxTime \wedge MinTime$$

Formula  $HCTime$  contains the variable  $t$ , and the specification of the real-time clock should describe only the changes to  $hr$  (the clock display) and  $now$  (the time). So, we have to hide  $t$ . Hiding is expressed in TLA<sup>+</sup> by the temporal existential quantifier  $\exists$ , introduced in Section 4.3 (page 41). However, as explained in that section, we can't simply write  $\exists t : HCTime$ . We must define  $HCTime$  in a module that declares  $t$ , and then use a parametrized instantiation of that module. This is done in Figure 9.1 on page 121. Instead of defining  $HCTime$  in a completely separate module, I have defined it in a submodule named *Inner* of the module *RealTimeHourClock* containing the specification of the real-time hour clock. Note that all the symbols declared and defined in the main module

up to that point can be used in the submodule. Submodule *Inner* is instantiated in the main module with the statement

$$I(t) \triangleq \text{INSTANCE } \textit{Inner}$$

The  $t$  in *HCTime* can then be hidden by writing  $\exists t : I(t)!\text{HCTime}$ .

The formula  $HC \wedge (\exists t : I(t)!\text{HCTime})$  describes the possible changes to the value of *hr*, and relates those changes to the value of *now*. But it says very little about how the value of *now* can change. For example, it allows the following behavior:

$$\begin{bmatrix} hr = 11 \\ now = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 11 \\ now = 23.4 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 11 \\ now = 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr = 11 \\ now = 23.4 \end{bmatrix} \rightarrow \dots$$

Because time can't go backwards, such a behavior doesn't represent a physical possibility. Everyone knows that time only increases, so there's no need to forbid this behavior if the only purpose of our specification is to describe the hour clock. However, a specification should also allow us to reason about a system. If the clock ticks approximately once per hour, then it can't stop. However, as the behavior above shows, the formula  $HC \wedge (\exists t : I(t)!\text{HCTime})$  by itself allows the clock to stop. To infer that it can't, we also need to state how *now* changes.

We define a formula *RTnow* that specifies the possible changes to *now*. This formula does not specify the granularity of the changes to *now*; it allows a step to advance *now* by a microsecond or by a century. However, we have decided that a step that changes *hr* should leave *now* unchanged, which implies that a step that changes *now* should leave *hr* unchanged. Therefore, steps that change *now* are described by the following action, where *Real* is the set of all real numbers:

$$\begin{aligned} \textit{NowNext} \triangleq & \wedge \textit{now}' \in \{r \in \textit{Real} : r > \textit{now}\} \quad \textit{now}' \text{ can equal any real number } > \textit{now} \\ & \wedge \text{UNCHANGED } \textit{hr} \end{aligned}$$

Formula *RTnow* should also allow steps that leave *now* unchanged. The initial value of *now* is an arbitrary real number (we can start the system at any time), so the safety part of *RTnow* is

$$(\textit{now} \in \textit{Real}) \wedge \square[\textit{NowNext}]_{\textit{now}}$$

The liveness condition we want is that *now* should increase without bound. Simple weak fairness of the *NowNext* action isn't good enough, because it allows "Zeno" behaviors such as

$$[\textit{now} = .9] \rightarrow [\textit{now} = .99] \rightarrow [\textit{now} = .999] \rightarrow [\textit{now} = .9999] \rightarrow \dots$$

in which the value of *now* remains bounded. Weak fairness of the action  $\textit{NowNext} \wedge (\textit{now}' > r)$  implies that eventually a *NowNext* step will occur in which the new value of *now* is greater than *r*. (This action is always enabled, so weak fairness implies that infinitely many such actions must occur.) Asserting

Weak fairness is discussed in Chapter 8.

MODULE *RealTimeHourClock*EXTENDS *Reals*, *HourClock*VARIABLE *now* The current time, measured in seconds.CONSTANT *Rho* A positive real number.ASSUME  $(Rho \in Real) \wedge (Rho > 0)$ MODULE *Inner*VARIABLE *t* $TNext \triangleq t' = \text{IF } HC_{nxt} \text{ THEN } 0 \text{ ELSE } t + (now' - now)$  $Timer \triangleq (t = 0) \wedge \square[TNext]_{(t, hr, now)}$  *t* is the elapsed time since the last *HC<sub>nxt</sub>* step. $MaxTime \triangleq \square(t \leq 3600 + Rho)$ *t* is always at most  $3600 + Rho$ . $MinTime \triangleq \square[HC_{nxt} \Rightarrow t \geq 3600 - Rho]_{hr}$  An *HC<sub>nxt</sub>* step can occur only if  $t \geq 3600 - Rho$ . $HCTime \triangleq Timer \wedge MaxTime \wedge MinTime$  $I(t) \triangleq \text{INSTANCE } Inner$  $NowNext \triangleq \wedge now' \in \{r \in Real : r > now\} \wedge \text{UNCHANGED } hr$  A *NowNext* step can advance *now* by any amount while leaving *hr* unchanged. $RTnow \triangleq \wedge now \in Real \wedge \square[NowNext]_{now} \wedge \forall r \in Real : \text{WF}_{now}(NowNext \wedge (now' > r))$  *RTnow* specifies how time may change. $RTHC \triangleq HC \wedge RTnow \wedge (\exists t : I(t) ! HCTime)$  The complete specification.**Figure 9.1:** The real-time specification of an hour clock that ticks every hour, plus or minus *Rho* seconds.

this for all real numbers *r* implies that *now* grows without bound, so we take as the fairness condition<sup>1</sup>

$$\forall r \in Real : \text{WF}_{now}(NowNext \wedge (now' > r))$$

The complete specification *RTHC* of the real-time hour clock, with the definition of formula *RTnow*, is in the *RealTimeHourClock* module of Figure 9.1 on this page. That module extends the standard *Reals* module, which defines the set *Real* of real numbers.

<sup>1</sup>An equivalent condition is  $\forall r \in Real : \diamondsuit(now > r)$ , but I like to express fairness with WF and SF formulas.

## 9.2 Real-Time Specifications in General

In Section 8.4 (page 96), we saw that the appropriate generalization of the liveness requirement that the hour clock tick infinitely often is weak fairness of the clock-tick action. There is a similar generalization for real-time specifications. Weak fairness of an action  $A$  asserts that if  $A$  is continuously enabled, then an  $A$  step must eventually occur. The real-time analog is that if  $A$  is continuously enabled for  $\epsilon$  seconds, then an  $A$  step must occur. Since an  $HCnxt$  action is always enabled, the requirement that the clock tick at least once every  $3600 + \rho$  seconds can be expressed in this way by letting  $A$  be  $HCnxt$  and  $\epsilon$  be  $3600 + \rho$ .

The requirement that an  $HCnxt$  action occur at most once every  $3600 - \rho$  seconds can be similarly generalized to the condition that an action  $A$  must be continuously enabled for at least  $\delta$  seconds before an  $A$  step can occur.

The first condition, the upper bound  $\epsilon$  on how long  $A$  can be enabled without an  $A$  step occurring, is vacuously satisfied if  $\epsilon$  equals *Infinity*—a value defined in the *Reals* module to be greater than any real number. The second condition, the lower bound  $\delta$  on how long  $A$  must be enabled before an  $A$  step can occur, is vacuously satisfied if  $\delta$  equals 0. So, nothing is lost by combining both of these conditions into a single formula containing  $\delta$  and  $\epsilon$  as parameters. I now define such a formula, which I call a *real-time bound condition*.

The weak fairness formula  $WF_v(A)$  actually asserts weak fairness of the action  $\langle A \rangle_v$ , which equals  $A \wedge (v' \neq v)$ . The subscript  $v$  is needed to rule out stuttering steps. Since the truth of a meaningful formula can't depend on whether or not there are stuttering steps, it makes no sense to say that an  $A$  step did or did not occur if that step could be a stuttering step. For this reason, the corresponding real-time condition must also be a condition on an action  $\langle A \rangle_v$ , not on an arbitrary action  $A$ . In most cases of interest,  $v$  is the tuple of all variables that occur in  $A$ . I therefore define the real-time bound formula  $RTBound(A, v, \delta, \epsilon)$  to assert that

- An  $\langle A \rangle_v$  step cannot occur until  $\langle A \rangle_v$  has been continuously enabled for at least  $\delta$  time units since the last  $\langle A \rangle_v$  step—or since the beginning of the behavior.
- $\langle A \rangle_v$  can be continuously enabled for at most  $\epsilon$  time units before an  $\langle A \rangle_v$  step occurs.

$RTBound(A, v, \delta, \epsilon)$  generalizes the formula  $\exists t : I(t)!HCTime$  of the real-time hour-clock specification, and it can be defined in the same way, using a submodule. However, the definition can be structured a little more compactly as

$$RTBound(A, v, D, E) \triangleq \text{LET } Timer(t) \triangleq \dots \\ \dots \\ \text{IN } \exists t : Timer(t) \wedge \dots$$

For the TLA<sup>+</sup> specification, I have replaced  $\delta$  and  $\epsilon$  by  $D$  and  $E$ .

We first define  $Timer(t)$  to be a temporal formula asserting that  $t$  always equals the length of time that  $\langle A \rangle_v$  has been continuously enabled since the last  $\langle A \rangle_v$  step. The value of  $t$  should be set to 0 by an  $\langle A \rangle_v$  step or a step that disables  $\langle A \rangle_v$ . A step that advances  $now$  should increment  $t$  by  $now' - now$  iff  $\langle A \rangle_v$  is enabled. Changes to  $t$  are therefore described by the action

$$\begin{aligned} TNext(t) \triangleq t' = & \text{ IF } \langle A \rangle_v \vee \neg(\text{ENABLED } \langle A \rangle_v)' \\ & \text{ THEN } 0 \\ & \text{ ELSE } t + (now' - now) \end{aligned}$$

We are interested in the meaning of  $Timer(t)$  only when  $v$  is a tuple whose components include all the variables that may appear in  $A$ . In this case, a step that leaves  $v$  unchanged cannot enable or disable  $\langle A \rangle_v$ . So, the formula  $Timer(t)$  should allow steps that leave  $t$ ,  $v$ , and  $now$  unchanged. Letting the initial value of  $t$  be 0, we define

$$Timer(t) \triangleq (t = 0) \wedge \square[TNext(t)]_{\langle t, v, now \rangle}$$

Formulas  $MaxTime$  and  $MinTime$  of the real-time hour clock's specification have the obvious generalizations:

- $MaxTime(t)$  asserts that  $t$  is always less than or equal to  $E$ :

$$MaxTime(t) \triangleq \square(t \leq E)$$

- $MinTime(t)$  asserts that an  $\langle A \rangle_v$  step can occur only if  $t \geq D$ :

$$MinTime(t) \triangleq \square[A \Rightarrow (t \geq D)]_v$$

(An equally plausible definition of  $MinTime(t)$  is  $\square[\langle A \rangle_v \Rightarrow (t \geq D)]_v$ , but the two are, in fact, equivalent.)

We then define  $RTBound(A, v, D, E)$  to equal

$$\exists t : Timer(t) \wedge MaxTime(t) \wedge MinTime(t)$$

We must also generalize formula  $RTnow$  of the real-time hour clock's specification. That formula describes how  $now$  changes, and it asserts that  $hr$  remains unchanged when  $now$  changes. The generalization is the formula  $RTnow(v)$ , which replaces  $hr$  with an arbitrary state function  $v$  that will usually be the tuple of all variables, other than  $now$ , appearing in the specification. Using these definitions, the specification  $RTHC$  of the real-time hour clock can be written

$$HC \wedge RTnow(hr) \wedge RTBound(HCnxt, hr, 3600 - Rho, 3600 + Rho)$$

The *RealTime* module, with its definitions of  $RTBound$  and  $RTnow$ , appears in Figure 9.2 on page 125.

Strong fairness strengthens weak fairness by requiring an  $A$  step to occur not just if action  $A$  is continuously enabled, but if it is repeatedly enabled. Being

repeatedly enabled includes the possibility that it is also repeatedly disabled. We can similarly strengthen our real-time bound conditions by defining a stronger formula  $SRTBound(A, v, \delta, \epsilon)$  to assert that

- An  $\langle A \rangle_v$  step cannot occur until  $\langle A \rangle_v$  has been enabled for a total of at least  $\delta$  time units since the last  $\langle A \rangle_v$  step—or since the beginning of the behavior.
- $\langle A \rangle_v$  can be enabled for a total of at most  $\epsilon$  time units before an  $\langle A \rangle_v$  step occurs.

If  $\epsilon < \text{Infinity}$ , then  $RTBound(A, v, \delta, \epsilon)$  implies that an  $\langle A \rangle_v$  step must occur if  $\langle A \rangle_v$  is continuously enabled for  $\epsilon$  seconds. Hence, if  $\langle A \rangle_v$  is ever enabled forever, infinitely many  $\langle A \rangle_v$  steps must occur. Thus,  $RTBound(A, v, \delta, \epsilon)$  implies weak fairness of  $A$ . More precisely,  $RTBound(A, v, \delta, \epsilon)$  and  $RTnow(v)$  together imply  $WF_v(A)$ . However,  $SRTBound(A, v, \delta, \epsilon)$  does not similarly imply strong fairness of  $A$ . It allows behaviors in which  $\langle A \rangle_v$  is enabled infinitely often but never executed—for example,  $A$  can be enabled for  $\epsilon/2$  seconds, then for  $\epsilon/4$  seconds, then for  $\epsilon/8$  seconds, and so on. For this reason,  $SRTBound$  does not seem to be of much practical use, so I won't bother defining it formally.

## 9.3 A Real-Time Caching Memory

Let's now use the *RealTime* module to write a real-time versions of the linearizable memory specification of Section 5.3 (page 51) and the write-through cache specification of Section 5.6 (page 54). We obtain the real-time memory specification by strengthening the specification in module *Memory* (Figure 5.3 on page 53) to require that the memory responds to a processor's requests within  $Rho$  seconds. The complete memory specification *Spec* of module *Memory* was obtained by hiding the variables *mem*, *ctl*, and *buf* in the internal specification *ISpec* of module *InternalMemory*. It's generally easier to add a real-time constraint to an internal specification, where the constraints can mention the internal (hidden) variables. So, we first add the timing constraint to *ISpec* and then hide the internal variables.

To specify that the system must respond to a processor request within  $Rho$  seconds, we add an upper-bound timing constraint for an action that becomes enabled when a request is issued, and that becomes disabled (possibly by being executed) only when the processor responds to the request. In specification *ISpec*, responding to a request requires two actions—*Do(p)* to perform the operation internally, and *Rsp(p)* to issue the response. Neither of these actions is the one we want; we have to define a new action for the purpose. There is a pending request for processor  $p$  iff *ctl[p]* equals “rdy”. So, we assert that the

MODULE *RealTime*

This module declares the variable *now*, which represents real time, and defines operators for writing real-time specifications. Real-time constraints are added to a specification by conjoining it with  $RTnow(v)$  and formulas of the form  $RTBound(A, v, \delta, \epsilon)$  for actions  $A$ , where  $v$  is the tuple of all specification variables and  $0 \leq \delta \leq \epsilon \leq \text{Infinity}$ .

EXTENDS *Reals*

VARIABLE *now* The value of *now* is a real number that represents the current time, in unspecified units.

$RTBound(A, v, \delta, \epsilon)$  asserts that an  $\langle A \rangle_v$  step can occur only after  $\langle A \rangle_v$  has been continuously enabled for  $\delta$  time units since the last  $\langle A \rangle_v$  step (or the beginning of the behavior), and it must occur before  $\langle A \rangle_v$  has been continuously enabled for more than  $\epsilon$  time units since the last  $\langle A \rangle_v$  step (or the beginning of the behavior).

$RTBound(A, v, D, E) \triangleq$

LET  $TNext(t) \triangleq t' = \text{IF } \langle A \rangle_v \vee \neg(\text{ENABLED } \langle A \rangle_v)' \text{ THEN } 0 \text{ ELSE } t + (now' - now)$

$Timer(t) \triangleq (t = 0) \wedge \square[TNext(t)]_{\langle t, v, now \rangle}$

$MaxTime(t) \triangleq \square(t \leq E)$  Asserts that  $t$  is always  $\leq E$ .

$MinTime(t) \triangleq \square[A \Rightarrow (t \geq D)]_v$  Asserts that an  $\langle A \rangle_v$  step can occur only if  $t \geq D$ .

IN  $\exists t : Timer(t) \wedge MaxTime(t) \wedge MinTime(t)$

$Timer(t)$  asserts that  $t$  is the length of time  $\langle A \rangle_v$  has been continuously enabled without an  $\langle A \rangle_v$  step occurring.

$RTnow(v)$  asserts that *now* is a real number that is increased without bound, in arbitrary increments, by steps that leave  $v$  unchanged.

$RTnow(v) \triangleq \text{LET } NowNext } \triangleq \wedge now' \in \{r \in \text{Real} : r > now\} \wedge \text{UNCHANGED } v$

IN  $\wedge now \in \text{Real}$   
 $\wedge \square[NowNext]_{now}$   
 $\wedge \forall r \in \text{Real} : \text{WF}_{now}(NowNext \wedge (now' > r))$

**Figure 9.2:** The *RealTime* module for writing real-time specifications.

following action cannot be enabled for more than *Rho* seconds without being executed:

$Respond(p) \triangleq (ctl[p] \neq \text{"rdy"}) \wedge (ctl'[p] = \text{"rdy"})$

The complete specification is formula *RTSpec* of module *RTMemory* in Figure 9.3 on the next page. To permit variables *mem*, *ctl*, and *buf* to be hidden, the *RTMemory* module contains a submodule *Inner* that extends module *InternalMemory*.

Having added a real-time constraint to the specification of a linearizable memory, let's strengthen the specification of the write-through cache so it sat-

---

MODULE *RTMemory*

A specification that strengthens the linearizable memory specification of Section 5.3 by requiring that a response be sent to every processor request within *Rho* seconds.

EXTENDS *MemoryInterface*, *RealTime*

CONSTANT *Rho*

ASSUME  $(Rho \in \text{Real}) \wedge (Rho > 0)$

---

MODULE *Inner*

We introduce a submodule so we can hide the variables *mem*, *ctl*, and *buf*.

EXTENDS *InternalMemory*

$\text{Respond}(p) \triangleq (ctl[p] \neq \text{"rdy"}) \wedge (ctl'[p] = \text{"rdy"})$  *Respond*(*p*) is enabled when a request is received from *p*; it is disabled when a *Respond*(*p*) step issues the response.

$\text{RTISpec} \triangleq \wedge \text{ISpec} \wedge \forall p \in \text{Proc} : \text{RTBound}(\text{Respond}(p), \text{ctl}, 0, \text{Rho}) \wedge \text{RTnow}(\langle \text{memInt}, \text{mem}, \text{ctl}, \text{buf} \rangle)$  We assert an upper-bound delay of *Rho* on *Respond*(*p*), for all processors *p*.

$\text{Inner}(\text{mem}, \text{ctl}, \text{buf}) \triangleq \text{INSTANCE Inner}$

$\text{RTSpec} \triangleq \exists \text{mem}, \text{ctl}, \text{buf} : \text{Inner}(\text{mem}, \text{ctl}, \text{buf})! \text{RTISpec}$

---

**Figure 9.3:** A real-time version of the linearizable memory specification.

isfies that constraint. The object is not just to add any real-time constraint that does the job—that’s easy to do by using the same constraint that we added to the memory specification. We want to write a specification of a real-time algorithm—a specification that tells an implementer how to meet the real-time constraints. This is generally done by placing real-time bounds on the original actions of the untimed specification, not by adding time bounds on a new action, as we did for the memory specification. An upper-bound constraint on the response time should be achieved by enforcing upper-bound constraints on the system’s actions.

If we try to achieve a bound on response time by adding real-time bounds to the write-through cache specification’s actions, we encounter the following problem. Operations by different processors “compete” with one another to enqueue operations on the finite queue *memQ*. For example, when servicing a write request for processor *p*, the system must execute a *DoWr*(*p*) action to enqueue the operation to the tail of *memQ*. That action is not enabled if *memQ* is full. The *DoWr*(*p*) action can be continually disabled by the system performing *DoWr* or *RdMiss* actions for other processors. That’s why, to guarantee liveness—that each request eventually receives a response—in Section 8.7 (page 107) we had to assert strong fairness of *DoWr* and *RdMiss* actions. The only way to ensure

that a  $DoWr(p)$  action is executed within some length of time is to use lower-bound constraints on the actions of other processors to ensure that they cannot perform  $DoWr$  or  $RdMiss$  actions too frequently. Although such a specification is possible, it is not the kind of approach anyone is likely to take in practice.

The usual method of enforcing real-time bounds on accesses to a shared resource is to schedule the use of the resource by different processors. So, let's modify the write-through cache to add a scheduling discipline to actions that enqueue operations on  $memQ$ . We use round-robin scheduling, which is probably the easiest one to implement. Suppose processors are numbered from 0 through  $N - 1$ . Round-robin scheduling means that an operation for processor  $p$  is the next one to be enqueued after an operation for processor  $q$  iff there is not an operation for any of the processors  $(q + 1) \% N, (q + 2) \% N, \dots, (p - 1) \% N$  waiting to be put on  $memQ$ .

To express this formally, we first let the set  $Proc$  of processors equal the set  $0 \dots (N - 1)$  of integers. We normally do this by defining  $Proc$  to equal  $0 \dots (N - 1)$ . However, we want to reuse the parameters and definitions from the write-through cache specification, and that's easiest to do by extending module  $WriteThroughCache$ . Since  $Proc$  is a parameter of that module, we can't define it. We therefore let  $N$  be a new constant parameter and let  $Proc = 0 \dots (N - 1)$  be an assumption.<sup>2</sup>

To implement round-robin scheduling, we use a variable  $lastP$  that equals the last processor whose operation was enqueued to  $memQ$ . We define the operator  $position$  so that  $p$  is the  $position(p)$ <sup>th</sup> processor after  $lastP$  in the round-robin order:

$$position(p) \triangleq \text{CHOOSE } i \in 1 \dots N : p = (lastP + i) \% N$$

(Thus,  $position(lastP)$  equals  $N$ .) An operation for processor  $p$  can be the next to access  $memQ$  iff there is no operation for a processor  $q$  with  $position(q) < position(p)$  ready to access it—that is, iff  $canGoNext(p)$  is true, where

$$canGoNext(p) \triangleq \forall q \in Proc : (position(q) < position(p)) \Rightarrow \neg \text{ENABLED } (RdMiss(q) \vee DoWr(q))$$

We then define  $RTRdMiss(p)$  and  $RTDoWr(p)$  to be the same as  $RdMiss(p)$  and  $DoWr(p)$ , respectively, except that they have the additional enabling condition  $canGoNext(p)$ , and they set  $lastP$  to  $p$ . The other subactions of the next-state action are the same as before, except that they must also leave  $lastP$  unchanged.

For simplicity, we assume a single upper bound of  $Epsilon$  on the length of time any of the actions of processor  $p$  can remain enabled without being executed—except for the  $Evict(p, a)$  action, which we never require to happen. In general, suppose  $A_1, \dots, A_k$  are actions such that (i) no two of them are

---

<sup>2</sup>We could also instantiate module  $WriteThroughCache$  with  $0 \dots (N - 1)$  substituted for  $Proc$ ; but that would require declaring the other parameters of  $WriteThroughCache$ , including the ones from the  $MemoryInterface$  module.

ever simultaneously enabled, and (ii) once any  $A_i$  becomes enabled, it must be executed before another  $A_j$  can be enabled. In this case, a single  $RTBound$  constraint on  $A_1 \vee \dots \vee A_k$  is equivalent to separate constraints on all the  $A_i$ . We can therefore place a single constraint on the disjunction of all the actions of processor  $p$ , except that we can't use the same constraint for both  $DoRd(p)$  and  $RTRdMiss(p)$  because an  $Evict(p, a)$  step could disable  $DoRd(p)$  and enable  $RTRdMiss(p)$ . We therefore use a separate constraint for  $RTRdMiss(p)$ .

We assume an upper bound of  $Delta$  on the time  $MemQWr$  or  $MemQRd$  can be enabled without dequeuing an operation from  $memQ$ . The variable  $memQ$  represents a physical queue between the bus and the main memory, and  $Delta$  must be large enough so an operation inserted into an empty queue will reach the memory and be dequeued within  $Delta$  seconds.

We want the real-time write-through cache to implement the real-time memory specification. This requires an assumption relating  $Delta$ ,  $Epsilon$ , and  $Rho$  to assure that the memory specification's timing constraint is satisfied—namely, that the delay between when the memory receives a request from processor  $p$  and when it responds is at most  $Rho$ . Determining this assumption requires computing an upper bound on that delay. Finding the smallest upper bound is hard; it's easier to show that

$$2 * (N + 1) * Epsilon + (N + QLen) * Delta$$

is an upper bound. So we assume that this value is less than or equal to  $Rho$ .

The complete specification appears in Figure 9.4 on the following two pages. The module also asserts as a theorem that the specification  $RTSpec$  of the real-time write-through cache implements (implies) the real-time memory specification, formula  $RTSpec$  of module  $RTMemory$ .

## 9.4 Zeno Specifications

I have described the formula  $RTBound(HCnxt, hr, \delta, \epsilon)$  as asserting that an  $HCnxt$  step must occur within  $\epsilon$  seconds of the previous  $HCnxt$  step. However, implicit in this description is a notion of causality that is not present in the formula. It would be just as accurate to describe the formula as asserting that *now* cannot advance by more than  $\epsilon$  seconds before the next  $HCnxt$  step occurs. The formula doesn't tell us whether this condition is met by causing the clock to tick or by preventing time from advancing. Indeed, the formula is satisfied by a “Zeno” behavior:<sup>3</sup>

$$\left[ \begin{array}{l} hr = 11 \\ now = 0 \end{array} \right] \rightarrow \left[ \begin{array}{l} hr = 11 \\ now = \epsilon/2 \end{array} \right] \rightarrow \left[ \begin{array}{l} hr = 11 \\ now = 3\epsilon/4 \end{array} \right] \rightarrow \left[ \begin{array}{l} hr = 11 \\ now = 7\epsilon/8 \end{array} \right] \rightarrow \dots$$

---

<sup>3</sup>The Greek philosopher Zeno posed the paradox that an arrow first had to travel half the distance to its target, then the next quarter of the distance, then the next eighth, and so on; thus it should not be able to land within a finite length of time.

MODULE *RTWriteThroughCache*EXTENDS *WriteThroughCache*, *RealTime*CONSTANT *N*We assume that the set *Proc* of processors equals  $0 \dots N - 1$ .ASSUME  $(N \in \text{Nat}) \wedge (Proc = 0 \dots N - 1)$ CONSTANTS *Delta*, *Epsilon*, *Rho* Some real-time bounds on actions.ASSUME  $\wedge (Delta \in \text{Real}) \wedge (Delta > 0)$   
 $\wedge (Epsilon \in \text{Real}) \wedge (Epsilon > 0)$   
 $\wedge (Rho \in \text{Real}) \wedge (Rho > 0)$   
 $\wedge 2 * (N + 1) * Epsilon + (N + QLen) * Delta \leq Rho$ We modify the write-through cache specification to require that operations for different processors are enqueued on *memQ* in round-robin order.VARIABLE *lastP* The last processor to enqueue an operation on *memQ*.*RTInit*  $\triangleq$  *Init*  $\wedge (lastP \in Proc)$  Initially, *lastP* can equal any processor.*position(p)*  $\triangleq$  *p* is the *position(p)*<sup>th</sup> processor after *lastP* in the round-robin order.CHOOSE  $i \in 1 \dots N : p = (lastP + i) \% N$ *canGoNext(p)*  $\triangleq$  True if processor *p* can be the next to enqueue an operation on *memQ*. $\forall q \in Proc : (position(q) < position(p)) \Rightarrow \neg \text{ENABLED} (RdMiss(q) \vee DoWr(q))$ *RTRdMiss(p)*  $\triangleq$   $\wedge canGoNext(p)$   
 $\wedge RdMiss(p)$   
 $\wedge lastP' = p$  Actions *RTRdMiss(p)* and *RTDoWr(p)* are the same as *RdMiss(p)* and *DoWr(p)* except that they are not enabled unless *p* is the next processor in the round-robin order ready to enqueue an operation on *memQ*, and they set *lastP* to *p*.*RTDoWr(p)*  $\triangleq$   $\wedge canGoNext(p)$   
 $\wedge DoWr(p)$   
 $\wedge lastP' = p$ *RTNext*  $\triangleq$   $\vee \exists p \in Proc : RTRdMiss(p) \vee RTDoWr(p)$   
 $\vee \wedge \vee \exists p \in Proc : \vee Req(p) \vee Rsp(p) \vee DoRd(p)$   
 $\vee \exists a \in Adr : Evict(p, a)$   
 $\vee MemQWr \vee MemQRd$   
 $\wedge \text{UNCHANGED } lastP$ The next-state action *RTNext* is the same as *Next* except with *RTRdMiss(p)* and *RTDoWr(p)* replaced by *RdMiss(p)* and *DoWr(p)*, and with other actions modified to leave *lastP* unchanged.*vars*  $\triangleq \langle memInt, wmem, buf, ctl, cache, memQ, lastP \rangle$ 

Figure 9.4a: A real-time version of the write-through cache (beginning).

$$\begin{aligned}
RTSpec &\triangleq \\
&\wedge RTInit \wedge \square[RTNext]_{vars} \\
&\wedge RTBound(MemQWr \vee MemQRd, vars, 0, Delta) \\
&\wedge \forall p \in Proc : \wedge RTBound(RTDoWr(p) \vee DoRd(p) \vee Rsp(p), \\
&\quad vars, 0, Epsilon) \\
&\quad \wedge RTBound(RTRdMiss(p), vars, 0, Epsilon) \\
&\wedge RTnow(vars)
\end{aligned}$$

We put an upper-bound delay of *Delta* on *MemQWr* and *MemQRd* actions (which dequeue operations from *memQ*), and an upper-bound delay of *Epsilon* on other actions.

$$RTM \triangleq \text{INSTANCE } RTMemory$$

THEOREM  $RTSpec \Rightarrow RTM!RTSpec$

**Figure 9.4b:** A real-time version of the write-through cache (end).

in which  $\epsilon$  seconds never pass. We rule out such Zeno behaviors by conjoining to our specification the formula  $RTnow(hr)$ —more precisely by conjoining its liveness conjunct

$$\forall r \in Real : \text{WF}_{now}(Next \wedge (now' > r))$$

which implies that time advances without bound. Let's call this formula *NZ* (for Non-Zeno).

Zeno behaviors pose no problem; they are trivially forbidden by conjoining *NZ*. A problem does exist if a specification allows *only* Zeno behaviors. For example, suppose we conjoined to the untimed hour-clock's specification the condition  $RTBound(HCnxt, hr, \delta, \epsilon)$  for some  $\delta$  and  $\epsilon$  with  $\delta > \epsilon$ . This would assert that the clock must wait at least  $\delta$  seconds before ticking, but must tick within a shorter length of time. In other words, the clock could never tick. Only a Zeno behavior, in which  $\epsilon$  seconds never elapsed, can satisfy this specification. Conjoining *NZ* to this specification yields a formula that allows no behaviors—that is, a formula equivalent to FALSE.

This example is an extreme case of what is called a *Zeno specification*. A Zeno specification is one for which there exists a finite behavior  $\sigma$  that satisfies the safety part but cannot be extended to an infinite behavior that satisfies both the safety part and *NZ*.<sup>4</sup> In other words, the only complete behaviors satisfying the safety part that extend  $\sigma$  are Zeno behaviors. A specification that is not Zeno is, naturally enough, said to be *non-Zeno*. By the definition of machine closure (in Section 8.9.2 on page 111), a specification is non-Zeno iff it is machine closed. More precisely, it is non-Zeno iff the pair of properties consisting of the safety part of the specification (the conjunction of the untimed specification, the real-time bound conditions, and the safety part of the *RTnow* formula) and *NZ* is machine closed.

<sup>4</sup>Recall that, on page 112, a finite behavior  $\sigma$  was defined to satisfy a safety property  $P$  iff adding infinitely many stuttering steps to the end of  $\sigma$  produces a behavior that satisfies  $P$ .

A Zeno specification is one in which the requirement that time increases without bound rules out some finite behaviors that would otherwise be allowed. Such a specification is likely to be incorrect because the real-time bound conditions are probably constraining the system in unintended ways. In this respect, Zeno specifications are much like other non-machine-closed specifications.

Section 8.9.2 mentions that the conjunction of fairness conditions on subactions of the next-state relation produces a machine closed specification. There is an analogous result for *RTBound* conditions and non-Zeno specifications. A specification is non-Zeno if it is the conjunction of (i) a formula of the form  $Init \wedge \square[Next]_{vars}$ , (ii) the formula  $RTnow(vars)$ , and (iii) a finite number of formulas of the form  $RTBound(A_i, vars, \delta_i, \epsilon_i)$ , where for each  $i$

- $0 \leq \delta_i \leq \epsilon_i \leq Infinity$
- $A_i$  is a subaction of the next-state action *Next*.
- No step is both an  $A_i$  and an  $A_j$  step, for any  $A_j$  with  $j \neq i$ .

The definition of a subaction appears on page 111.

In particular, this implies that the specification *RTSpec* of the real-time write-through cache in module *RTWriteThroughCache* is non-Zeno.

This result does not apply to the specification of the real-time memory in module *RTMemory* (Figure 9.3 on page 126) because the action *Respond(p)* is not a subaction of the next-state action *INext* of formula *ISpec*. The specification is nonetheless non-Zeno, because any finite behavior  $\sigma$  that satisfies the specification can be extended to one in which time advances without bound. For example, we can first extend  $\sigma$  to respond to all pending requests immediately (in 0 time), and then extend it to an infinite behavior by adding steps that just increase *now*.

*INext* is defined on page 53

It's easy to construct an example in which conjoining an *RTBound* formula for an action that is not a subaction of the next-state action produces a Zeno specification. For example, consider the formula

$$(9.2) \quad HC \wedge RTBound(hr' = hr - 1, hr, 0, 3600) \wedge RTnow(hr)$$

where *HC* is the specification of the hour clock. The next-state action *HCnxt* of *HC* asserts that *hr* is either incremented by 1 or changes from 12 to 1. The *RTBound* formula asserts that *now* cannot advance for 3600 or more seconds without an  $hr' = hr - 1$  step occurring. Since *HC* asserts that every step that changes *hr* is an *HCnxt* step, the safety part of (9.2) is satisfied only by behaviors in which *now* increases by less than 3600 seconds. Since the complete specification (9.2) contains the conjunct *NZ*, which asserts that *now* increases without bound, it is equivalent to FALSE, and is thus a Zeno specification.

When a specification describes how a system is implemented, the real-time constraints are likely to be expressed as *RTBound* formulas for subactions of the next-state action. These are the kinds of formulas that correspond fairly directly to an implementation. For example, module *RTWriteThroughCache*

describes an algorithm for implementing a memory, and it has real-time bounds on subactions of the next-state action. On the other hand, more abstract, higher-level specifications—ones describing what a system is supposed to do rather than how to do it—are less likely to have real-time constraints expressed in this way. Thus, the high-level specification of the real-time memory in module *RTMemory* contains an *RTBound* formula for an action that is not a subaction of the next-state action.

## 9.5 Hybrid System Specifications

A system described by a TLA<sup>+</sup> specification is a physical entity. The specification's variables represent some part of the physical state—the display of a clock, or the distribution of charge in a piece of silicon that implements a memory cell. In a real-time specification, the variable *now* is different from the others because we are not abstracting away the continuous nature of time. The specification allows *now* to assume any of a continuum of values. The discrete states in a behavior mean that we are observing the state of the system, and hence the value of *now*, at a sequence of discrete instants.

There may be physical quantities other than time whose continuous nature we want to represent in a specification. For an air traffic control system, we might want to represent the positions and velocities of the aircraft. For a system controlling a nuclear reactor, we might want to represent the physical parameters of the reactor itself. A specification that represents such continuously varying quantities is called a *hybrid system specification*.

As an example, consider a system that, among other things, controls a switch that influences the one-dimensional motion of some object. Suppose the object's position *p* obeys one of the following laws, depending on whether the switch is off or on:

$$(9.3) \quad \begin{aligned} d^2p/dt^2 + c * dp/dt + f[t] &= 0 \\ d^2p/dt^2 + c * dp/dt + f[t] + k * p &= 0 \end{aligned}$$

where *c* and *k* are constants, *f* is some function, and *t* represents time. At any instant, the future position of the object is determined by the object's current position and velocity. So, the state of the object is described by two variables—namely, its position *p* and its velocity *w*. These variables are related by *w* = *dp/dt*.

We describe this system with a TLA<sup>+</sup> specification in which the variables *p* and *w* are changed only by steps that change *now*—that is, steps representing the passage of time. We specify the changes to the discrete system state and any real-time constraints as before. However, we replace *RTnow(v)* with a formula having the following next-state action, where *Integrate* and *D* are explained

below, and  $v$  is the tuple of all discrete variables:

$$\begin{aligned} & \wedge \text{now}' \in \{r \in \text{Real} : r > \text{now}\} \\ & \wedge \langle p', w' \rangle = \text{Integrate}(D, \text{now}, \text{now}', \langle p, w \rangle) \\ & \wedge \text{UNCHANGED } v \quad \text{The discrete variables change instantaneously.} \end{aligned}$$

The second conjunct asserts that  $p'$  and  $w'$  equal the expressions obtained by solving the appropriate differential equation for the object's position and velocity at time  $\text{now}'$ , assuming that their values at time  $\text{now}$  are  $p$  and  $w$ . The differential equation is specified by  $D$ , while *Integrate* is a general operator for solving (integrating) an arbitrary differential equation.

To specify the differential equation satisfied by the object, let's suppose that *switchOn* is a Boolean-valued state variable that describes the position of the switch. We can then rewrite the pair of equations (9.3) as

$$d^2p/dt^2 + c * dp/dt + f[t] + (\text{IF } \text{switchOn} \text{ THEN } k * p \text{ ELSE } 0) = 0$$

We then define the function  $D$  so this equation can be written as

$$D[t, p, dp/dt, d^2p/dt^2] = 0$$

Using the TLA<sup>+</sup> notation for defining functions of multiple arguments, which is explained in Section 16.1.7 on page 301, the definition is

$$\begin{aligned} D[t, p0, p1, p2 \in \text{Real}] & \triangleq \\ & p2 + c * p1 + f[t] + (\text{IF } \text{switchOn} \text{ THEN } k * p0 \text{ ELSE } 0) \end{aligned}$$

We obtain the desired specification if the operator *Integrate* is defined so that  $\text{Integrate}(D, t_0, t_1, \langle x_0, \dots, x_{n-1} \rangle)$  is the value at time  $t_1$  of the  $n$ -tuple

$$\langle x, dx/dt, \dots, d^{n-1}/dt^{n-1} \rangle$$

where  $x$  is a solution to the differential equation

$$D[t, x, dx/dt, \dots, d^n x / dt^n] = 0$$

whose 0<sup>th</sup> through  $(n-1)^{\text{st}}$  derivatives at time  $t_0$  are  $x_0, \dots, x_{n-1}$ . The definition of *Integrate* appears in the *DifferentialEquations* module of Section 11.1.3 (page 174).

In general, a hybrid-system specification is similar to a real-time specification, except that the formula  $\text{RTnow}(v)$  is replaced by one that describes the changes to all variables that represent continuously changing physical quantities. The *Integrate* operator will allow you to specify those changes for many hybrid systems. Some systems will require different operators. For example, describing the evolution of some physical quantities might require an operator for describing the solution to a partial differential equation. However, if you can describe the evolution mathematically, then it can be specified in TLA<sup>+</sup>.

Hybrid system specifications still seem to be of only academic interest, so I won't say any more about them. If you do have occasion to write one, this brief discussion should indicate how you can do it.

## 9.6 Remarks on Real Time

Real-time constraints are used most often to place an upper bound on how long it can take the system to do something. In this capacity, they can be considered a strong form of liveness, specifying not just that something must eventually happen, but when it must happen. In very simple specifications, such as the hour clock and the write-through cache, real-time constraints usually replace liveness conditions. More complicated specifications can assert both real-time constraints and liveness properties.

The real-time specifications I have seen have not required very complicated timing constraints. They have been specifications either of fairly simple algorithms in which timing constraints are crucial to correctness, or of more complicated systems in which real time appears only through the use of simple timeouts to ensure liveness. I suspect that people don't build systems with complicated real-time constraints because it's too hard to get them right.

I've described how to write a real-time specification by conjoining *RTnow* and *RTBound* formulas to an untimed specification. One can prove that all real-time specifications can be written in this form. In fact, it suffices to use *RTBound* formulas only for subactions of the next-state action. However, this result is of theoretical interest only because the resulting specification can be incredibly complicated. The operators *RTnow* and *RTBound* solve all the real-time specification problems that I have encountered; but I haven't encountered enough to say with confidence that they're all you will ever need. Still, I am quite confident that, whatever real-time properties you have to specify, it will not be hard to express them in TLA<sup>+</sup>.

# Chapter 10

## Composing Specifications

Systems are usually described in terms of their components. In the specifications we've written so far, the components have been represented as separate disjuncts of the next-state action. For example, the FIFO system pictured on page 35 is specified in module *InnerFIFO* on page 38 by representing the three components with the following disjuncts of the next-state action:

Sender:  $\exists msg \in Message : SSend(msg)$

Buffer:  $BufRcv \vee BufSend$

Receiver:  $RRcv$

In this chapter, we learn how to specify the components separately and compose their specifications to form a single system specification. Most of the time, there's no point doing this. The two ways of writing the specification differ by only a few lines—a trivial difference in a specification of hundreds or thousands of lines. Still, you may encounter a situation in which it's better to specify a system as a composition.

First, we must understand what it means to compose specifications. We usually say that a TLA formula specifies the correct behavior of a system. However, as explained in Section 2.3 (page 18), a behavior actually represents a possible history of the entire universe, not just of the system. So, it would be more accurate to say that a TLA formula specifies a universe in which the system behaves correctly. Building a system that implements a specification  $F$  means constructing the universe so it satisfies  $F$ . (Fortunately, correctness of the system depends on the behavior of only a tiny part of the universe, and that's the only part we must build.) Composing two systems whose specifications are  $F$  and  $G$  means making the universe satisfy both  $F$  and  $G$ , which is the same as making it satisfy  $F \wedge G$ . Thus, the specification of the composition of two systems is the conjunction of their specifications.

Writing a specification as the composition of its components therefore means writing the specification as a conjunction, each conjunct of which can be viewed as the specification of a component. While the basic idea is simple, the details are not always obvious. To simplify the exposition, I begin by considering only safety properties, ignoring liveness and largely ignoring hiding. Liveness and hiding are discussed in Section 10.6.

## 10.1 Composing Two Specifications

Let's return once again to the simple hour clock, with no liveness or real-time requirement. In Chapter 2, we specified such a clock whose display is represented by the variable  $hr$ . We can write that specification as

$$(hr \in 1 \dots 12) \wedge \square[HGN(hr)]_{hr}$$

where  $HGN$  is defined by

$$HGN(h) \triangleq h' = (h \% 12) + 1$$

Now let's write a specification  $TwoClocks$  of a system composed of two separate hour clocks, whose displays are represented by the variables  $x$  and  $y$ . (The two clocks are not synchronized and are completely independent of one another.) We can just define  $TwoClocks$  to be the conjunction of the two clock specifications

$$\begin{aligned} TwoClocks \triangleq & \wedge (x \in 1 \dots 12) \wedge \square[HGN(x)]_x \\ & \wedge (y \in 1 \dots 12) \wedge \square[HGN(y)]_y \end{aligned}$$

The following calculation shows how we can rewrite  $TwoClocks$  in the usual form as a “monolithic” specification with a single next-state action:<sup>1</sup>

$$\begin{aligned} TwoClocks & \\ \equiv & \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \\ & \wedge \square[HGN(x)]_x \wedge \square[HGN(y)]_y \\ \equiv & \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \quad \text{Because } \square(F \wedge G) \equiv (\square F) \wedge (\square G). \\ & \wedge \square([HGN(x)]_x \wedge [HGN(y)]_y) \\ \equiv & \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \quad \text{By definition of } [\dots]_x \text{ and } [\dots]_y. \\ & \wedge \square(\wedge HGN(x) \vee x' = x \\ & \quad \wedge HGN(y) \vee y' = y) \end{aligned}$$

<sup>1</sup>This calculation is informal because it contains formulas that are not legal TLA—namely, ones of the form  $\square A$  where  $A$  is an action that doesn't have the syntactic form  $[B]_v$ . However, it can be done rigorously.

$$\begin{aligned}
&\equiv \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \\
&\quad \wedge \square (\vee HCN(x) \wedge HCN(y) \\
&\quad \quad \vee HCN(x) \wedge (y' = y) \\
&\quad \quad \vee HCN(y) \wedge (x' = x) \\
&\quad \quad \vee (x' = x) \wedge (y' = y) ) \\
&\equiv \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \\
&\quad \wedge \square [\vee HCN(x) \wedge HCN(y) \\
&\quad \quad \vee HCN(x) \wedge (y' = y) \\
&\quad \quad \vee HCN(y) \wedge (x' = x) ]_{\langle x, y \rangle}
\end{aligned}$$

Because: 
$$\begin{pmatrix} \wedge \vee A_1 \\ \vee A_2 \\ \wedge \vee B_1 \\ \vee B_2 \end{pmatrix} \equiv \begin{pmatrix} \vee A_1 \wedge B_1 \\ \vee A_2 \wedge B_1 \\ \vee A_2 \wedge B_2 \end{pmatrix}$$

By definition of  $[\dots]_{\langle x, y \rangle}$ .

Thus, *TwoClocks* is equivalent to  $Init \wedge \square[TCNxt]_{\langle x, y \rangle}$  where the next-state action *TCNxt* is

$$\begin{aligned}
TCNxt &\triangleq \vee HCN(x) \wedge HCN(y) \\
&\quad \vee HCN(x) \wedge (y' = y) \\
&\quad \vee HCN(y) \wedge (x' = x)
\end{aligned}$$

This next-state action differs from the ones we are used to writing because of the disjunct  $HCN(x) \wedge HCN(y)$ , which represents the simultaneous advance of the two displays. In the specifications we have written so far, different components never act simultaneously.

Up until now, we have been writing what are called *interleaving* specifications. In an interleaving specification, each step represents an operation of only one component. For example, in our FIFO specification, a (nonstuttering) step represents an action of either the sender, the buffer, or the receiver. For want of a better term, we describe as *noninterleaving* a specification that, like *TwoClocks*, does permit simultaneous actions by two components.

Suppose we want to write an interleaving specification of the two-clock system as the conjunction of two component specifications. One way is to replace the next-state actions  $HCN(x)$  and  $HCN(y)$  of the two components by two actions  $HCNx$  and  $HCNy$  so that, when we perform the analogous calculation to the one above, we get

$$\begin{pmatrix} \wedge (x \in 1 \dots 12) \wedge \square[HCNx]_x \\ \wedge (y \in 1 \dots 12) \wedge \square[HCNy]_y \end{pmatrix} \equiv \begin{pmatrix} \wedge (x \in 1 \dots 12) \wedge (y \in 1 \dots 12) \\ \wedge \square [\vee HCNx \wedge (y' = y) \\ \quad \quad \vee HCNy \wedge (x' = x) ]_{\langle x, y \rangle} \end{pmatrix}$$

From the calculation above, we see that this equivalence holds if the following three conditions are satisfied: (i)  $HCNx$  implies  $HCN(x)$ , (ii)  $HCNy$  implies  $HCN(y)$ , and (iii)  $HCNx \wedge HCNy$  implies  $x' = x$  or  $y' = y$ . (Condition (iii) implies that the disjunct  $HCNx \wedge HCNy$  of the next-state action is subsumed by one of the disjuncts  $HCNx \wedge (y' = y)$  and  $HCNy \wedge (x' = x)$ .) The common way

of satisfying these conditions is to let the next-state action of each clock assert that the other clock's display is unchanged. We do this by defining

$$HCNx \triangleq HCN(x) \wedge (y' = y) \quad HCNy \triangleq HCN(y) \wedge (x' = x)$$

Another way to write an interleaving specification is simply to disallow simultaneous changes to both clock displays. We can do this by taking as our specification the formula

$$TwoClocks \wedge \square[(x' = x) \vee (y' = y)]_{(x, y)}$$

The second conjunct asserts that any step must leave  $x$  or  $y$  (or both) unchanged.

Everything we have done for the two-clock system generalizes to any system comprising two components. The same calculation as above shows that if

$$(v_1' = v_1) \wedge (v_2' = v_2) \equiv (v' = v) \quad \text{This asserts that } v \text{ is unchanged iff both } v_1 \text{ and } v_2 \text{ are.}$$

then

$$(10.1) \quad \left( \wedge I_1 \wedge \square[N_1]_{v_1} \wedge I_2 \wedge \square[N_2]_{v_2} \right) \equiv \left( \begin{array}{l} \wedge I_1 \wedge I_2 \\ \wedge \square[\vee N_1 \wedge N_2 \\ \quad \vee N_1 \wedge (v_2' = v_2) \\ \quad \vee N_2 \wedge (v_1' = v_1)]_v \end{array} \right)$$

for any state predicates  $I_1$  and  $I_2$  and any actions  $N_1$  and  $N_2$ . The left-hand side of this equivalence represents the composition of two component specifications if  $v_k$  is a tuple containing the variables that describe the  $k^{\text{th}}$  component, for  $k = 1, 2$ , and  $v$  is the tuple of all the variables.

The equivalent formulas in (10.1) represent an interleaving specification if the first disjunct in the next-state action of the right-hand side is redundant, so it can be removed. This is the case if  $N_1 \wedge N_2$  implies that  $v_1$  or  $v_2$  is unchanged. The usual way to ensure that this condition is satisfied is by defining each  $N_k$  so it implies that the other component's tuple is left unchanged. Another way to obtain an interleaving specification is by conjoining the formula  $\square[(v_1' = v_1) \vee (v_2' = v_2)]_v$ .

## 10.2 Composing Many Specifications

We can generalize (10.1) to the composition of any set  $C$  of components. Because universal quantification generalizes conjunction, the following rule is a generalization of (10.1):

**Composition Rule** For any set  $C$ , if

$$(\forall k \in C : v_k' = v_k) \equiv (v' = v) \quad \text{This asserts that } v \text{ is unchanged iff all the } v_k \text{ are.}$$

then

$$\begin{aligned}
 (\forall k \in C : I_k \wedge \square[N_k]_{v_k}) &\equiv \\
 &\wedge \forall k \in C : I_k \\
 &\wedge \square \left[ \vee \exists k \in C : N_k \wedge (\forall i \in C \setminus \{k\} : v_i' = v_i) \right. \\
 &\quad \left. \wedge \square \left[ \vee \exists i, j \in C : (i \neq j) \wedge N_i \wedge N_j \wedge F_{ij} \right]_v \right]
 \end{aligned}$$

for some actions  $F_{ij}$ .

The second disjunct of the next-state action is redundant, and we have an interleaving specification, if each  $N_i$  implies that  $v_j$  is unchanged, for all  $j \neq i$ . However, for this to hold,  $N_i$  must mention  $v_j$  for components  $j$  other than  $i$ . You might object to this approach—either on philosophical grounds, because you feel that the specification of one component should not mention the state of another component, or because mentioning other component’s variables complicates the component’s specification. An alternative approach is simply to assert interleaving. You can do this by conjoining the following formula, which states that no step changes both  $v_i$  and  $v_j$ , for any  $i$  and  $j$  with  $i \neq j$ :

$$\square [\exists k \in C : \forall i \in C \setminus \{k\} : v_i' = v_i]_v$$

This conjunct can be viewed as a global condition, not attached to any component’s specification.

For the left-hand side of the conclusion of the Composition Rule to represent the composition of separate components, the  $v_k$  need not be composed of separate variables. They could contain different “parts” of the same variable that describe different components. For example, our system might consist of a set  $Clock$  of separate, independent clocks, where clock  $k$ ’s display is described by the value of  $hr[k]$ . Then  $v_k$  would equal  $hr[k]$ . It’s easy to specify such an array of clocks as a composition. Using the definition of  $HCN$  on page 136 above, we can write the specification as

$$(10.2) \ ClockArray \triangleq \forall k \in Clock : (hr[k] \in 1 \dots 12) \wedge \square[HCN(hr[k])]_{hr[k]}$$

This is a noninterleaving specification, since it allows simultaneous steps by different clocks.

Suppose we wanted to use the Composition Rule to express  $ClockArray$  as a monolithic specification. What would we substitute for  $v$ ? Our first thought is to substitute  $hr$  for  $v$ . However, the hypothesis of the rule requires that  $v$  must be left unchanged iff  $hr[k]$  is left unchanged, for all  $k \in Clock$ . However, as explained in Section 6.5 on page 72, specifying the values of  $hr[k']$  for all  $k \in Clock$  does not specify the value of  $hr$ . It doesn’t even imply that  $hr$  is a function. We must substitute for  $v$  the function  $hrfcn$  defined by

$$(10.3) \ hrfcn \triangleq [k \in Clock \mapsto hr[k]]$$

The function  $hrcn$  equals  $hr$  iff  $hr$  is a function with domain  $Clock$ . Formula  $ClockArray$  does not imply that  $hr$  is always a function. It specifies the possible values of  $hr[k]$ , for all  $k \in Clock$ , but it doesn't specify the value of  $hr$ . Even if we changed the initial condition to imply that  $hr$  is initially a function with domain  $Clock$ , formula  $ClockArray$  would not imply that  $hr$  is always a function. For example, it would still allow “stuttering” steps that leave each  $hr[k]$  unchanged, but change  $hr$  in unknown ways.

We might prefer to write a specification in which  $hr$  is a function with domain  $Clock$ . One way of doing this is to conjoin to the specification the formula  $\square IsFcnOn(hr, Clock)$ , where  $IsFcnOn(hr, Clock)$  asserts that  $hr$  is an arbitrary function with domain  $Clock$ . The operator  $IsFcnOn$  is defined by

$$IsFcnOn(f, S) \triangleq f = [x \in S \mapsto f[x]]$$

We can view the formula  $\square IsFcnOn(hr, Clock)$  as a global constraint on  $hr$ , while the value of  $hr[k]$  for each component  $k$  is described by that component's specification.

Now, suppose we want to write an interleaving specification of the array of clocks as the composition of specifications of the individual clocks. In general, the conjunction in the Composition Rule is an interleaving specification if each  $N_k$  implies that  $v_i$  is unchanged, for all  $i \neq k$ . So, we want the next-state action  $N_k$  of clock  $k$  to imply that  $hr[i]$  is unchanged for every clock  $i$  other than  $k$ . The most obvious way to do this is to define  $N_k$  to equal

$$\begin{aligned} & \wedge hr'[k] = (hr[k] \% 12) + 1 \\ & \wedge \forall i \in Clock \setminus \{k\} : hr'[i] = hr[i] \end{aligned}$$

We can express this formula more compactly using the EXCEPT construct. This construct applies only to functions, so we must choose whether or not to require  $hr$  to be a function. If  $hr$  is a function, then we can let  $N_k$  equal

$$(10.4) \quad hr' = [hr \text{ EXCEPT } ![k] = (hr[k] \% 12) + 1]$$

As noted above, we can ensure that  $hr$  is a function by conjoining the formula  $\square IsFcnOn(hr, Clock)$  to the specification. Another way is to define the state function  $hrcn$  by (10.3) on the preceding page and let  $N(k)$  equal

$$hrcn' = [hrcn \text{ EXCEPT } ![k] = (hr[k] \% 12) + 1]$$

A specification is just a mathematical formula; as we've seen before, there are often many equivalent ways of writing a formula. Which one you choose is usually a matter of taste.

The EXCEPT construct is explained in Section 5.2 on page 48.

## 10.3 The FIFO

Let's now specify the FIFO, described in Chapter 4, as the composition of its three components—the Sender, the Buffer, and the Receiver. We start with the

internal specification, in which the variable  $q$  occurs—that is,  $q$  is not hidden. First, we decide what part of the state describes each component. The variables  $in$  and  $out$  are channels. Recall that the *Channel* module (page 30) specifies a channel *chan* to be a record with *val*, *rdy*, and *ack* components. The *Send* action, which sends a value, modifies the *val* and *rdy* components; the *Rcv* action, which receives a value, modifies the *ack* component. So, the components' states are described by the following state functions:

Sender:  $\langle in.val, in.rdy \rangle$

Buffer:  $\langle in.ack, q, out.val, out.rdy \rangle$

Receiver:  $out.ack$

Unfortunately, we can't reuse the definitions from the *InnerFIFO* module on page 38 for the following reason. The variable  $q$ , which is hidden in the final specification, is part of the Buffer component's internal state. Therefore, it should not appear in the specifications of the Sender or Receiver component. The Sender and Receiver actions defined in the *InnerFIFO* module all mention  $q$ , so we can't use them. We therefore won't bother reusing that module. However, instead of starting completely from scratch, we can make use of the *Send* and *Rcv* actions from the *Channel* module on page 30 to describe the changes to *in* and *out*.

Let's write a noninterleaving specification. The next-state actions of the components are then the same as the corresponding disjuncts of the *Next* action in module *InnerFIFO*, except that they do not mention the parts of the states belonging to the other components. These contain *Send* and *Rcv* actions, instantiated from the *Channel* module, which use the EXCEPT construct. As noted above, we can apply EXCEPT only to functions—and to records, which are functions. We therefore add to our specification the conjunct

$$\square (IsChannel(in) \wedge IsChannel(out))$$

Section 5.2 on page 48 explains why records are functions.

where *IsChannel*( $c$ ) asserts that  $c$  is a channel—that is a record with *val*, *ack*, and *rdy* fields. Since a record with *val*, *ack*, and *rdy* fields is a function whose domain is {“val”, “ack”, “rdy”}, we can define *IsChannel*( $c$ ) to equal *IsFcnOn*( $c$ , {“val”, “ack”, “rdy”}). However, it's just as easy to define formula *IsChannel*( $c$ ) directly by

$$IsChannel(c) \triangleq c = [ack \mapsto c.ack, val \mapsto c.val, rdy \mapsto c.rdy]$$

In writing this specification, we face the same problem as in our original FIFO specification of introducing the variable  $q$  and then hiding it. In Chapter 4, we solved this problem by introducing  $q$  in a separate *InnerFIFO* module, which is instantiated by the *FIFO* module that defines the final specification. We do essentially the same thing here, except that we introduce  $q$  in a submodule

instead of in a completely separate module. All the symbols declared and defined at the point where the submodule appears can be used within it. The submodule itself can be instantiated in the containing module anywhere after it appears. (Submodules are used in the *RealTimeHourClock* and *RTMemory* specifications on pages 121 and 126 of Chapter 9.)

There is one small problem to be solved before we can write a composite specification of the FIFO—how to specify the initial predicates. It makes sense for the initial predicate of each component’s specification to specify the initial values of its part of the state. However the initial condition includes the requirements  $in.ack = in.rdy$  and  $out.ack = out.rdy$ , each of which relates the initial states of two different components. (These requirements are stated in module *InnerFIFO* by the conjuncts *InChan!Init* and *OutChan!Init* of the initial predicate *Init*.) There are three ways of expressing a requirement that relates the initial states of multiple components:

- Assert it in the initial conditions of all the components. Although symmetric, this seems needlessly redundant.
- Arbitrarily assign the requirement to one of the components. This intuitively suggests that we are assigning to that component the responsibility of ensuring that the requirement is met.
- Assert the requirement as a conjunct separate from either of the component specifications. This intuitively suggests that it is an assumption about how the components are put together, rather than a requirement of either component.

When we write an open-system specification, as described in Section 10.7 below, the intuitive suggestions of the last two approaches can be turned into formal requirements. I’ve taken the last approach and added

$$(in.ack = in.rdy) \wedge (out.ack = out.rdy)$$

as a separate condition. The complete specification is in module *CompositeFIFO* of Figure 10.1 on the next page. Formula *Spec* of this module is a noninterleaving specification; for example, it allows a single step that is both an *InChan!Send* step (the sender sends a value) and an *OutChan!Rcv* step (the receiver acknowledges a value). Hence, it is not equivalent to the interleaving specification *Spec* of the *FIFO* module on page 41, which does not allow such a step.

## 10.4 Composition with Shared State

Thus far, we have been considering *disjoint-state compositions*—ones in which the components are represented by disjoint parts of the state, and a compo-

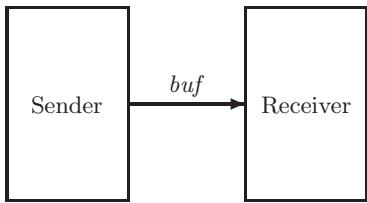
| MODULE <i>CompositeFIFO</i>                                                                                                |                                                             |
|----------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| EXTENDS <i>Naturals, Sequences</i>                                                                                         |                                                             |
| CONSTANT <i>Message</i>                                                                                                    |                                                             |
| VARIABLES <i>in, out</i>                                                                                                   |                                                             |
| $InChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow in$                     |                                                             |
| $OutChan \triangleq \text{INSTANCE } Channel \text{ WITH } Data \leftarrow Message, chan \leftarrow out$                   |                                                             |
| $SenderInit \triangleq (in.rdy \in \{0, 1\}) \wedge (in.val \in Message)$                                                  |                                                             |
| $Sender \triangleq SenderInit \wedge \square[\exists msg \in Message : InChan!Send(msg)]_{\langle in.val, in.rdy \rangle}$ |                                                             |
| MODULE <i>InnerBuf</i>                                                                                                     |                                                             |
| VARIABLE <i>q</i>                                                                                                          |                                                             |
| $BufferInit \triangleq \wedge in.ack \in \{0, 1\}$                                                                         |                                                             |
| $\wedge q = \langle \rangle$                                                                                               |                                                             |
| $\wedge (out.rdy \in \{0, 1\}) \wedge (out.val \in Message)$                                                               | The Buffer's internal specification, with <i>q</i> visible. |
| $BufRcv \triangleq \wedge InChan!Rcv$                                                                                      |                                                             |
| $\wedge q' = Append(q, in.val)$                                                                                            |                                                             |
| $\wedge \text{UNCHANGED } \langle out.val, out.rdy \rangle$                                                                |                                                             |
| $BufSend \triangleq \wedge q \neq \langle \rangle$                                                                         |                                                             |
| $\wedge OutChan!Send(Head(q))$                                                                                             |                                                             |
| $\wedge q' = Tail(q)$                                                                                                      |                                                             |
| $\wedge \text{UNCHANGED } in.ack$                                                                                          |                                                             |
| $InnerBuffer \triangleq BufferInit \wedge \square[BufRcv \vee BufSend]_{\langle in.ack, q, out.val, out.rdy \rangle}$      |                                                             |
| $Buf(q) \triangleq \text{INSTANCE } InnerBuf$                                                                              | The Buffer's external specification with <i>q</i> hidden.   |
| $Buffer \triangleq \exists q : Buf(q)!InnerBuffer$                                                                         |                                                             |
| $ReceiverInit \triangleq out.ack \in \{0, 1\}$                                                                             | The Receiver's specification.                               |
| $Receiver \triangleq ReceiverInit \wedge \square[OutChan!Rcv]_{out.ack}$                                                   |                                                             |
| $IsChannel(c) \triangleq c = [ack \mapsto c.ack, val \mapsto c.val, rdy \mapsto c.rdy]$                                    |                                                             |
| $Spec \triangleq \wedge \square(IsChannel(in) \wedge IsChannel(out))$                                                      | Asserts that <i>in</i> and <i>out</i> are always records.   |
| $\wedge (in.ack = in.rdy) \wedge (out.ack = out.rdy)$                                                                      | Relates different components' initial states.               |
| $\wedge Sender \wedge Buffer \wedge Receiver$                                                                              | Conjoins the three specifications.                          |

Figure 10.1: A noninterleaving composite specification of the FIFO.

ment's next-state action describes changes only to its part of the state.<sup>2</sup> We now consider the case when this may not be possible.

### 10.4.1 Explicit State Changes

We first examine the situation in which some part of the state cannot be partitioned among the different components, but the state change that each component performs is completely described by the specification. As an example, let's again consider a Sender and a Receiver that communicate with a FIFO buffer. In the system we studied in Chapter 4, sending or receiving a value required two steps. For example, the Sender executes a *Send* step to send a value, and it must then wait until the buffer executes a *Rcv* step before it can send another value. We simplify the system by replacing the Buffer component with a variable *buf* whose value is the sequence of values sent by the Sender but not yet received by the Receiver. This replaces the three-component system pictured on page 35 with this two-component one:



The Sender sends a value by appending it to the end of *buf*; the Receiver receives a value by removing it from the head of *buf*.

In general, the Sender performs some computation to produce the values that it sends, and the Receiver does some computation on the values that it receives. The system state consists of *buf* and two tuples *s* and *r* of variables that describe the Sender and Receiver states. In a monolithic specification, the system's next-state action is a disjunction *Sndr*  $\vee$  *Rcvr*, where *Sndr* and *Rcvr* describe steps taken by the Sender and Receiver, respectively. These actions are defined by

$$\begin{array}{ll}
 \begin{array}{l}
 \textit{Sndr} \triangleq \\
 \vee \wedge \textit{buf}' = \textit{Append}(\textit{buf}, \dots) \\
 \wedge \textit{SComm} \\
 \wedge \text{UNCHANGED } \textit{r} \\
 \vee \wedge \textit{SCompute} \\
 \wedge \text{UNCHANGED } \langle \textit{buf}, \textit{r} \rangle
 \end{array} &
 \begin{array}{l}
 \textit{Rcvr} \triangleq \\
 \vee \wedge \textit{buf} \neq \langle \rangle \\
 \wedge \textit{buf}' = \textit{Tail}(\textit{buf}) \\
 \wedge \textit{RComm} \\
 \wedge \text{UNCHANGED } \textit{s} \\
 \vee \wedge \textit{RCompute} \\
 \wedge \text{UNCHANGED } \langle \textit{buf}, \textit{s} \rangle
 \end{array}
 \end{array}$$

<sup>2</sup>In an interleaving composition, a component specification may assert that the state of other components is *not* changed.

for some actions  $SComm$ ,  $SCompute$ ,  $RComm$ , and  $RCompute$ . For simplicity, we assume that neither  $Sndr$  nor  $Rcvr$  allows stuttering actions, so  $SCompute$  changes  $s$  and  $RCompute$  changes  $r$ . We now write the specification as the composition of separate specifications of the Sender and Receiver.

Splitting the initial predicate is straightforward. The initial conditions on  $s$  belong to the Sender's initial predicate; those on  $r$  belong to the Receiver's initial predicate; and the initial condition  $buf = \langle \rangle$  can be assigned arbitrarily to either of them.

Now let's consider the next-state actions  $NS$  and  $NR$  of the Sender and Receiver components. The trick is to define them by

$$NS \triangleq Sndr \vee (\sigma \wedge (s' = s)) \quad NR \triangleq Rcvr \vee (\rho \wedge (r' = r))$$

where  $\sigma$  and  $\rho$  are actions containing only the variable  $buf$ . Think of  $\sigma$  as describing possible changes to  $buf$  that are not caused by the Sender, and  $\rho$  as describing possible changes to  $buf$  that are not caused by the Receiver. Thus,  $NS$  permits any step that is either a  $Sndr$  step or one that leaves  $s$  unchanged and is a change to  $buf$  that can't be "blamed" on the Sender.

Suppose  $\sigma$  and  $\rho$  satisfy the following three conditions:

- $\forall d : (buf' = Append(buf, d)) \Rightarrow \rho$

A step that appends a value to  $buf$  is not caused by the Receiver.

- $(buf \neq \langle \rangle) \wedge (buf' = Tail(buf)) \Rightarrow \sigma$

A step that removes a value from the head of  $buf$  is not caused by the Sender.

- $(\sigma \wedge \rho) \Rightarrow (buf' = buf)$

A step that is caused by neither the Sender nor the Receiver cannot change  $buf$ .

Using obvious relations such as<sup>3</sup>

$$(buf' = buf) \wedge (buf \neq \langle \rangle) \wedge (buf' = Tail(buf)) \equiv \text{FALSE}$$

a computation like the one by which we derived (10.1) shows

$$\square[NS]_{\langle buf, s \rangle} \wedge \square[NR]_{\langle buf, r \rangle} \equiv \square[Sndr \vee Rcvr]_{\langle buf, s, r \rangle}$$

Thus,  $NS$  and  $NR$  are suitable next-state actions for the components, if we choose  $\sigma$  and  $\rho$  to satisfy the three conditions above. There is considerable freedom in that choice. The strongest possible choices of  $\sigma$  and  $\rho$  are ones that describe exactly the changes permitted by the other component:

$$\sigma \triangleq (buf \neq \langle \rangle) \wedge (buf' = Tail(buf))$$

$$\rho \triangleq \exists d : buf' = Append(buf, d)$$

---

<sup>3</sup>These relations are true only if  $buf$  is a sequence. A rigorous calculation requires the use of an invariant to assert that  $buf$  actually is a sequence.

We can weaken these definitions any way we want, so long as we maintain the condition that  $\sigma \wedge \rho$  implies that  $buf$  is unchanged. For example, we can define  $\sigma$  as above and let  $\rho$  equal  $\neg\sigma$ . The choice is a matter of taste.

I've been describing an interleaving specification of the Sender/Receiver system. Now let's consider a noninterleaving specification—one that allows steps in which both the Sender and the Receiver are computing. In other words, we want the specification to allow  $SCompute \wedge RCompute$  steps that leave  $buf$  unchanged. Let  $SSndr$  be the action that is the same as  $Sndr$  except it doesn't mention  $r$ , and let  $RRcvr$  be defined analogously. We then have

$$Sndr \equiv SSndr \wedge (r' = r) \quad Rcvr \equiv RRcvr \wedge (s' = s)$$

A monolithic noninterleaving specification has the next-state action

$$Sndr \vee Rcvr \vee (SSndr \wedge RRcvr \wedge (buf' = buf))$$

It is the conjunction of component specifications having the next-state actions  $NS$  and  $NR$  defined by

$$NS \triangleq SSndr \vee (\sigma \wedge (s' = s)) \quad NR \triangleq RRcvr \vee (\rho \wedge (r' = r))$$

where  $\sigma$  and  $\rho$  are as above.

This two-process situation generalizes to the composition of any set  $C$  of components that share a variable or tuple of variables  $w$ . The interleaving case generalizes to the following rule, in which  $N_k$  is the next-state action of component  $k$ , the action  $\mu_k$  describes all changes to  $w$  that are attributed to some component other than  $k$ , the tuple  $v_k$  describes the private state of  $k$ , and  $v$  is the tuple formed by all the  $v_k$ :

#### Shared-State Composition Rule

The four conditions

$$1. (\forall k \in C : v_k' = v_k) \equiv (v' = v)$$

$v$  is unchanged iff the private state  $v_k$  of every component is unchanged.

$$2. \forall i, k \in C : N_k \wedge (i \neq k) \Rightarrow (v_i' = v_i)$$

The next-state action of any component  $k$  leaves the private state  $v_i$  of all other components  $i$  unchanged.

$$3. \forall i, k \in C : N_k \wedge (w' \neq w) \wedge (i \neq k) \Rightarrow \mu_i$$

A step of any component  $k$  that changes  $w$  is a  $\mu_i$  step, for any other component  $i$ .

$$4. (\forall k \in C : \mu_k) \equiv (w' = w)$$

A step is caused by no component iff it does not change  $w$ .

imply

$$(\forall k \in C : I_k \wedge \square[N_k \vee (\mu_k \wedge (v_k' = v_k))]_{\langle w, v_k \rangle})$$

$$\equiv (\forall k \in C : I_k) \wedge \square[\exists k \in C : N_k]_{\langle w, v \rangle}$$

Assumption 2 asserts that we have an interleaving specification. If we drop that assumption, then the right-hand side of the conclusion may not be a sensible specification, since a disjunct  $N_k$  may allow steps in which a variable of some other component assumes arbitrary values. However, if each  $N_k$  correctly determines the new values of component  $k$ 's private state  $v_k$ , then the left-hand side will be a reasonable specification, though possibly a noninterleaving one (and not necessarily equivalent to the right-hand side).

### 10.4.2 Composition with Joint Actions

Consider the linearizable memory of Chapter 5. As shown in the picture on page 45, it is a system consisting of a collection of processors, a memory, and an interface represented by the variable  $memInt$ . We now take it to be a two-component system, where the set of processors forms one component, called the *environment*, and the memory is the other component. Let's neglect hiding for now and consider only the internal specification, with all variables visible. We want to write the specification in the form

$$(10.5) \quad (IE \wedge \square[NE]_{vE}) \wedge (IM \wedge \square[NM]_{vM})$$

where  $E$  refers to the environment component (the processors) and  $M$  to the memory component. The tuple  $vE$  of variables includes  $memInt$  and the variables of the environment component; the tuple  $vM$  includes  $memInt$  and the variables of the memory component. We must choose the formulas  $IE$ ,  $NE$ , etc. so that (10.5), with internal variables hidden, is equivalent to the memory specification *Spec* of module *Memory* on page 53.

In the memory specification, communication between the environment and the memory is described by an action of the form

$$Send(p, d, memInt, memInt') \quad \text{or} \quad Reply(p, d, memInt, memInt')$$

where *Send* and *Reply* are unspecified operators declared in the *MemoryInterface* module (page 48). The specification says nothing about the actual value of  $memInt$ . So, not only do we not know how to split  $memInt$  into two parts that are each changed by only one of the components, we don't even know exactly how  $memInt$  changes.

The trick to writing the specification as a composition is to put the *Send* and *Reply* actions in the next-state actions of both components. We represent the sending of a value over  $memInt$  as a *joint action* performed by both the memory and the environment. The next-state actions have the following form:

$$NM \triangleq \exists p \in Proc : MRqst(p) \vee MRsp(p) \vee MInternal(p)$$

$$NE \triangleq \exists p \in Proc : ERqst(p) \vee ERsp(p)$$

where an  $MRqst(p) \wedge ERqst(p)$  step represents the sending of a request by processor  $p$  (part of the environment) to the memory, an  $MRsp(p) \wedge ERsp(p)$  step represents the sending of a reply by the memory to processor  $p$ , and an  $MInternal(p)$  step is an internal step of the memory component that performs the request. (There are no internal steps of the environment.)

The sending of a reply is controlled by the memory, which chooses what value is sent and when it is sent. The enabling condition and the value sent are therefore specified by the  $MRsp(p)$  action. Let's take the internal variables of the memory component to be the same variables  $mem$ ,  $ctl$ , and  $buf$  as in the internal monolithic memory specification of module *InternalMemory* on pages 52 and 53. We can then let  $MRsp(p)$  be the same as the action  $Rsp(p)$  defined in that module. The  $ERsp(p)$  action should always be enabled, and it should allow any legal response to be sent. A legal response is an element of  $Val$  or the special value  $NoVal$ , so we can define  $ERsp(p)$  to equal<sup>4</sup>

$$\begin{aligned} & \wedge \exists rsp \in Val \cup \{NoVal\} : Reply(p, rsp, memInt, memInt') \\ & \wedge \dots \end{aligned}$$

where the “ $\dots$ ” describes the new values of the environment's internal variables.

The sending of a request is controlled by the environment, which chooses what value is sent and when it is sent. Hence, the enabling condition should be part of the  $ERqst(p)$  action. In the monolithic specification of the *InternalMemory* module, that enabling condition was  $ctl[p] = "rdy"$ . However, if  $ctl$  is an internal variable of the memory, it can't also appear in the environment specification. We therefore have to add a new variable whose value indicates whether a processor is allowed to send a new request. Let's use a Boolean variable  $rdy$ , where  $rdy[p]$  is true iff processor  $p$  can send a request. The value of  $rdy[p]$  is set false when  $p$  sends a request and is set true again when the corresponding response to  $p$  is sent. We can therefore define  $ERqst(p)$ , and complete the definition of  $ERsp(p)$ , as follows:

$$\begin{aligned} ERqst(p) & \triangleq \wedge rdy[p] \\ & \wedge \exists req \in MReq : Send(p, req, memInt, memInt') \\ & \wedge rdy' = [rdy \text{ EXCEPT } ![p] = \text{FALSE}] \\ ERsp(p) & \triangleq \wedge \exists rsp \in Val \cup \{NoVal\} : \\ & \quad Reply(p, rsp, memInt, memInt') \\ & \quad \wedge rdy' = [rdy \text{ EXCEPT } ![p] = \text{TRUE}] \end{aligned}$$

The memory's  $MRqst(p)$  action is the same as the  $Req(p)$  action of the *InternalMemory* module, except without the enabling condition  $ctl[p] = "rdy"$ .

---

<sup>4</sup>The bound on the  $\exists$  isn't necessary. We can let the processor accept any value, not just a legal one, by taking  $\exists rsp : Reply(p, rsp, memInt, memInt')$  as the first conjunct. However, it's generally better to use bounded quantifiers when possible.

Finally, the memory's internal action  $MInternal(p)$  is the same as the  $Do(p)$  action of the *InternalMemory* module.

The rest of the specification is easy. The tuples  $vE$  and  $vM$  are  $\langle memInt, rdy \rangle$  and  $\langle memInt, mem, ctl, buf \rangle$ , respectively. Defining the initial predicates  $IE$  and  $IM$  is straightforward, except for the decision of where to put the initial condition  $memInt \in InitMemInt$ . We can put it in either  $IE$  or  $IM$ , in both, or else in a separate conjunct that belongs to neither component's specification. Let's put it in  $IM$ , which then equals the initial predicate  $IInit$  from the *InternalMemory* module. The final environment specification is obtained by hiding  $rdy$  in its internal specification; the final memory component specification is obtained by hiding  $mem$ ,  $ctl$ , and  $buf$  in its internal specification. The complete specification appears in Figure 10.2 on the next page. I have not bothered to define  $IM$ ,  $MRsp(p)$ , or  $MInternal(p)$ , since they equal  $IInit$ ,  $Rsp(p)$ , and  $Do(p)$  from the *InternalMemory* module, respectively.

What we've just done for the environment-memory system generalizes naturally to joint-action specifications of any two-component system in which part of the state cannot be considered to belong to either component. It also generalizes to systems in which any number of components share some part of the state. For example, suppose we want to write a composite specification of the linearizable memory system in which each processor is a separate component. The specification of the memory component would be the same as before. The next-state action of processor  $p$  would now be

$$ERqst(p) \vee ERsp(p) \vee OtherProc(p)$$

where  $ERqst(p)$  and  $ERsp(p)$  are the same as above, and an  $OtherProc(p)$  step represents the sending of a request by, or a response to, some processor other than  $p$ . Action  $OtherProc(p)$  represents  $p$ 's participation in the joint action by which another processor  $q$  communicates with the memory component. It is defined to equal

$$\begin{aligned} \exists q \in Proc \setminus \{p\} : & \vee \exists req \in MReq : Send(q, req, memInt, memInt') \\ & \vee \exists rsp \in Val \cup \{NoVal\} : \\ & \quad Reply(q, rsp, memInt, memInt') \end{aligned}$$

This example is rather silly because each processor must participate in communication actions that concern only other components. It would be better to change the interface to make  $memInt$  an array, with communication between processor  $p$  and the memory represented by a change to  $memInt[p]$ . A sensible example would require that a joint action represent a true interaction between all the components—for example, a barrier synchronization operation in which the components wait until they are all ready and then perform a synchronization step together.

```

 ━━━━━━━━━━━ MODULE JointActionMemory ━━━━━━━
 EXTENDS MemoryInterface
 ━━━━━━━ MODULE InnerEnvironmentComponent ━━━━━
 VARIABLE rdy
 IE \triangleq rdy = [p \in Proc \mapsto TRUE]
 ERqst(p) \triangleq \wedge rdy[p]
 \wedge \exists req \in MReq : Send(p, req, memInt, memInt)
 \wedge rdy' = [rdy EXCEPT ![p] = FALSE]
 ERsp(p) \triangleq \wedge \exists rsp \in Val \cup {NoVal} : Reply(p, rsp, memInt, memInt)
 \wedge rdy' = [rdy EXCEPT ![p] = TRUE]
 NE \triangleq \exists p \in Proc : ERqst(p) \vee ERsp(p)
 IESpec \triangleq IE \wedge \square [NE](memInt, rdy)
 ━━━━━━━━━━━ MODULE InnerMemoryComponent ━━━━━
 EXTENDS InternalMemory
 MRqst(p) \triangleq \wedge \exists req \in MReq : \wedge Send(p, req, memInt, memInt)
 \wedge buf' = [buf EXCEPT ![p] = req]
 \wedge ctl' = [ctl EXCEPT ![p] = “busy”]
 \wedge UNCHANGED mem
 NM \triangleq \exists p \in Proc : MRqst(p) \vee Do(p) \vee Rsp(p)
 IMSpec \triangleq IInit \wedge \square [NM](memInt, mem, ctl, buf)
 IEnv(rdy) \triangleq INSTANCE InnerEnvironmentComponent
 IMem(mem, ctl, buf) \triangleq INSTANCE InnerMemoryComponent
 Spec \triangleq \wedge \exists rdy : IEnv(rdy)!IESpec
 \wedge \exists mem, ctl, buf : IMem(mem, ctl, buf)!IMSpec
 ━

```

**Figure 10.2:** A joint-action specification of a linearizable memory.

## 10.5 A Brief Review

The basic idea of composing specifications is simple: a composite specification is the conjunction of formulas, each of which can be considered to be the specification of a separate component. This chapter has presented several techniques for writing a specification as a composition. Before going further, let’s put these techniques in perspective.

### 10.5.1 A Taxonomy of Composition

We have seen three different ways of categorizing composite specifications:

**Interleaving versus noninterleaving.** An interleaving specification is one in which each (nonstuttering) step can be attributed to exactly one component. A noninterleaving specification allows steps that represent simultaneous operations of two or more different components.

The word *interleaving* is standard; there is no common terminology for the other concepts.

**Disjoint-state versus shared-state.** A disjoint-state specification is one in which the state can be partitioned, with each part belonging to a separate component. A part of the state can be a variable  $v$ , or a “piece” of that variable such as  $v.c$  or  $v[c]$  for some fixed  $c$ . Any change to a component’s part of the state is attributed to that component. In a shared-state specification, some part of the state can be changed by steps attributed to more than one component.

**Joint-action versus separate-action.** A joint-action specification is a noninterleaving one in which some step attributed to one component must occur simultaneously with a step attributed to another component. A separate-action specification is simply one that is not a joint-action specification.

These are independent ways of classifying specifications, except that a joint-action specification must be noninterleaving.

### 10.5.2 Interleaving Reconsidered

Should we write interleaving or noninterleaving specifications? We might try to answer this question by asking: can different components really take simultaneous steps? However, this question makes no sense. A step is a mathematical abstraction; real components perform operations that take a finite amount of time. Operations performed by two different components could overlap in time. We are free to represent this physical situation either with a single simultaneous step of the two components, or with two separate steps. In the latter case, the specification usually allows the two steps to occur in either order. (If the two operations must occur simultaneously, then we have written a joint-action specification.) It’s up to you whether to write an interleaving or a noninterleaving specification. You should choose whichever is more convenient.

The choice is not completely arbitrary if you want one specification to implement another. A noninterleaving specification will not, in general, implement an interleaving one because the noninterleaving specification will allow simultaneous actions that the interleaving specification prohibits. So, if you want to write a lower-level specification that implements a higher-level interleaving specification, then you’ll have to use an interleaving specification. As we’ve seen, it’s easy

to turn a noninterleaving specification into an interleaving one by conjoining an interleaving assumption.

### 10.5.3 Joint Actions Reconsidered

The reason for writing a composite specification is to separate the specifications of the different components. The mixing of actions from different components in a joint-action specification destroys this separation. So, why should we write such a specification?

Joint-action specifications arise most often in highly abstract descriptions of inter-component communication. In writing a composite specification of the linearizable memory, we were led to use joint actions because of the abstract nature of the interface. In real systems, communication occurs when one component changes the state and another component later observes that change. The interface described by the *MemoryInterface* module abstracts away those two steps, replacing them with a single one that represents instantaneous communication—a fiction that does not exist in the real world. Since each component must remember that the communication has occurred, the single communication step has to change the private state of both components. That’s why we couldn’t use the approach of Section 10.4.1 (page 144), which requires that any change to the shared interface change the nonshared state of just one component.

The abstract memory interface simplifies the specification, allowing communication to be represented as one step instead of two. But this simplification comes at the cost of blurring the distinction between the two components. If we blur this distinction, it may not make sense to write the specification as the conjunction of separate component specifications. As the memory system example illustrates, decomposing the system into separate components communicating with joint actions may require the introduction of extra variables. There may occasionally be a good reason for adding this kind of complexity to a specification, but it should not be done as a matter of course.

## 10.6 Liveness and Hiding

### 10.6.1 Liveness and Machine Closure

Thus far, the discussion of composition has neglected liveness. In composite specifications, it is usually easy to specifying liveness by placing fairness conditions on the actions of individual components. For example, to specify an array of clocks that all keep ticking forever, we would modify the specification

*ClockArray* of (10.2) on page 139 to equal

$\forall k \in \text{Clock} :$

$$(hr[k] \in 1 \dots 12) \wedge \square[\text{HCN}(hr[k])]_{hr[k]} \wedge \text{WF}_{hr[k]}(\text{HCN}(hr[k]))$$

When writing a weak or strong fairness formula for an action  $A$  of a component  $c$ , there arises the question of what the subscript should be. The obvious choices are (i) the tuple  $v$  describing the entire specification state, and (ii) the tuple  $v_c$  describing that component's state. The choice can matter only if the safety part of the specification allows the system to reach some state in which an  $A$  step could leave  $v_c$  unchanged while changing  $v$ . Although unlikely, this could conceivably be the case in a joint-action specification. If it is, we probably don't want the fairness condition to be satisfied by a step that leaves the component's state unchanged, so we would use the subscript  $v_c$ .

Fairness conditions for composite specifications do raise one important question: if each component specification is machine closed, is the composite specification necessarily machine closed? Suppose we write the specification as  $\forall k \in C : S_k \wedge L_k$ , where each pair  $S_k, L_k$  is machine closed. Let  $S$  be the conjunction of the  $S_k$  and  $L$  the conjunction of the  $L_k$ , so the specification equals  $S \wedge L$ . The conjunction of safety properties is a safety property,<sup>5</sup> so  $S$  is a safety property. Hence, we can ask if the pair  $S, L$  is machine closed.

Machine closure is defined in Section 8.9.2 on page 111.

In general,  $S, L$  need not be machine closed. But, for an interleaving composition, it usually is. Liveness properties are usually expressed as the conjunction of weak and strong fairness properties of actions. As stated on page 112, a specification is machine closed if its liveness property is the conjunction of fairness properties for subactions of the next-state action. In an interleaving composition, each  $S_k$  usually has the form  $I_k \wedge \square[N_k]_{v_k}$  where the  $v_k$  satisfy the hypothesis of the Composition Rule (page 138), and each  $N_k$  implies  $v_i' = v_i$ , for all  $i$  in  $C \setminus \{k\}$ . In this case, the Composition Rule implies that a subaction of  $N_k$  is also a subaction of the next-state action of  $S$ . Hence, if we write an interleaving composition in the usual way, and we write machine-closed component specifications in the usual way, then the composite specification is machine closed.

It is not so easy to obtain a machine-closed noninterleaving composition—especially with a joint-action composition. We have actually seen an example of a joint-action specification in which each component is machine closed but the composition is not. In Chapter 9, we wrote a real-time specification by conjoining one or more *RTBound* formulas and an *RTnow* formula to an untimed specification. A pathological example was the following, which is formula (9.2) on page 131:

$$HC \wedge \text{RTBound}(hr' = hr - 1, hr, 0, 3600) \wedge \text{RTnow}(hr)$$

*HC* is the hour-clock specification from Chapter 2.

<sup>5</sup>Recall that a safety property is one that is violated by a behavior iff it is violated at some particular point in the behavior. A behavior violates a conjunction of safety properties  $S_k$  iff it violates some particular  $S_k$ , and that  $S_k$  is violated at some specific point.

We can view this formula as the conjunction of three component specifications:

1.  $HC$  specifies a clock, represented by the variable  $hr$ .
2.  $RTBound(hr' = hr - 1, hr, 0, 3600)$  specifies a timer, represented by the hidden (existentially quantified) timer variable.
3.  $RTnow(hr)$  specifies real time, represented by the variable  $now$ .

The formula is a joint-action composition, with two kinds of joint actions:

- Joint actions of the first and second components that change both  $hr$  and the timer.
- Joint actions of the second and third components that change both the timer and  $now$ .

The first two specifications are trivially machine closed because they assert no liveness condition, so their liveness property is TRUE. The third specification's safety property asserts that  $now$  is a real number that is changed only by steps that increment it and leave  $hr$  unchanged; its liveness property  $NZ$  asserts that  $now$  increases without bound. Any finite behavior satisfying the safety property can easily be extended to an infinite behavior satisfying the entire specification, so the third specification is also machine closed. However, as we observed in Section 9.4, the composite specification is Zeno, meaning that it's not machine closed.

## 10.6.2 Hiding

Suppose we can write a specification  $S$  as the composition of two component specifications  $S_1$  and  $S_2$ . Can we write  $\exists h : S$ , the specification  $S$  with variable  $h$  hidden, as a composition—that is, as the conjunction of two separate component specifications? If  $h$  represents state that is accessed by both components, then the answer is no. If the two components communicate through some part of the state, then that part of the state cannot be made internal to the separate components.

The simplest situation in which  $h$  doesn't represent shared state is when it occurs in only one of the component specifications—say,  $S_2$ . If  $h$  doesn't occur in  $S_1$ , then the equivalence

$$(\exists h : S_1 \wedge S_2) \equiv S_1 \wedge (\exists h : S_2)$$

provides the desired decomposition.

Now suppose that  $h$  occurs in both component specifications, but does not represent state accessed by both components. This can be the case only if different “parts” of  $h$  occur in the two component specifications. For example,

$h$  might be a record with components  $h.c1$  and  $h.c2$ , where  $S_1$  mentions only  $h.c1$  and  $S_2$  mentions only  $h.c2$ . In this case, we have

$$(\exists h : S_1 \wedge S_2) \equiv (\exists h1 : T_1) \wedge (\exists h2 : T_2)$$

where  $T_1$  is obtained from  $S_1$  by substituting the variable  $h1$  for the expression  $h.c1$ , and  $T_2$  is defined similarly. Of course we can use any variables in place of  $h1$  and  $h2$ ; in particular, we can replace them both by the same variable.

We can generalize this result as follows to the composition of any finite number<sup>6</sup> of components:

**Compositional Hiding Rule** If the variable  $h$  does not occur in the formula  $T_i$ , and  $S_i$  is obtained from  $T_i$  by substituting  $h[i]$  for  $q$ , then

$$(\exists h : \forall i \in C : S_i) \equiv (\forall i \in C : \exists q : T_i)$$

for any finite set  $C$ .

The assumption that  $h$  does not occur in  $T_i$  means that the variable  $h$  occurs in formula  $S_i$  only in the expression  $h[i]$ . This in turn implies that the composition  $\forall i \in C : S_i$  does not determine the value of  $h$ , just of its components  $h[i]$  for  $i \in C$ . As noted in Section 10.2 on page 138, we can make the composite specification determine the value of  $h$  by conjoining the formula  $\square \text{IsFcnOn}(h, C)$  to it, where  $\text{IsFcnOn}$  is defined on page 140. The hypotheses of the Compositional Hiding Rule imply

$$(\exists h : \square \text{IsFcnOn}(h, C) \wedge \forall i \in C : S_i) \equiv (\forall i \in C : \exists q : T_i)$$

Now consider the common case in which  $\forall i \in C : S_i$  is an interleaving composition, where each specification  $S_i$  describes changes to  $h[i]$  and asserts that steps of component  $i$  leave  $h[j]$  unchanged for  $j \neq i$ . We cannot apply the Compositional Hiding Rule because  $S_i$  must mention other components of  $h$  besides  $h[i]$ . For example, it probably contains an expression of the form

$$(10.6) \quad h' = [h \text{ EXCEPT } !(i) = \text{exp}]$$

which mentions all of  $h$ . However, we can transform  $S_i$  into a specification  $\widehat{S}_i$  that describes only the changes to  $h[i]$  and makes no assertions about other components. For example, we can replace (10.6) with  $h'[i] = \text{exp}$ , and we can replace an assertion that  $h$  is unchanged by the assertion that  $h[i]$  is unchanged. The composition  $\forall i \in C : \widehat{S}_i$  may allow steps that change two different components  $h[i]$  and  $h[j]$ , while leaving all other variables unchanged, making it a noninterleaving specification. It will then not be equivalent to  $\forall i \in C : S_i$ , which requires that the changes to  $h[i]$  and  $h[j]$  be performed by different steps. However, it can be shown that hiding  $h$  hides this difference, making the two specifications equivalent. We can then apply the Compositional Hiding Rule with  $S_i$  replaced by  $\widehat{S}_i$ .

<sup>6</sup>The Compositional Hiding Rule is not true in general if  $C$  is an infinite set; but the examples in which it doesn't hold are pathological and don't arise in practice.

## 10.7 Open-System Specifications

A specification describes the interaction between a system and its environment. For example, the FIFO buffer specification of Chapter 4 specifies the interaction between the buffer (the system) and an environment consisting of the sender and receiver. So far, all the specifications we have written have been complete-system specifications, meaning that they are satisfied by a behavior that represents the correct operation of both the system and its environment. When we write such a specification as the composition of an environment specification  $E$  and a system specification  $M$ , it has the form  $E \wedge M$ .

An open-system specification is one that can serve as a contract between a user of the system and its implementer. An obvious choice of such a specification is the formula  $M$  that describes the correct behavior of the system component by itself. However, such a specification is unimplementable. It asserts that the system acts correctly no matter what the environment does. A system cannot behave as expected in the face of arbitrary behavior of its environment. It would be impossible to build a buffer that satisfies the buffer component's specification regardless of what the sender and receiver did. For example, if the sender sends a value before the previous value has been acknowledged, then the buffer could read the value while it is changing, causing unpredictable results.

A contract between a user and an implementer should require the system to act correctly only if the environment does. If  $M$  describes correct behavior of the system and  $E$  describes correct behavior of the environment, such a specification should require that  $M$  be true if  $E$  is. This suggests that we take as our open-system specification the formula  $E \Rightarrow M$ , which is true if the system behaves correctly or the environment behaves incorrectly. However,  $E \Rightarrow M$  is too weak a specification for the following reason. Consider again the example of a FIFO buffer, where  $M$  describes the buffer and  $E$  the sender and receiver. Suppose now that the buffer sends a new value before the receiver has acknowledged the previous one. This could cause the receiver to act incorrectly, possibly modifying the output channel in some way not allowed by the receiver's specification. This situation is described by a behavior in which both  $E$  and  $M$  are false—a behavior that satisfies the specification  $E \Rightarrow M$ . However, the buffer should not be considered to act correctly in this case, since it was the buffer's error that caused the receiver to act incorrectly. Hence, this behavior should not satisfy the buffer's specification.

An open-system specification should assert that the system behaves correctly at least as long as the environment does. To express this, we introduce a new temporal operator  $\dagger\Rightarrow$ , where  $E \dagger\Rightarrow M$  asserts that  $M$  remains true at least one step longer than  $E$  does, remaining true forever if  $E$  does. Somewhat more precisely,  $E \dagger\Rightarrow M$  asserts that

- $E$  implies  $M$ .

Open-system specifications are sometimes called *rely-guarantee* or *assume-guarantee* specifications.

- If the safety property of  $E$  is not violated by the first  $n$  states of a behavior, then the safety property of  $M$  is not violated by the first  $n+1$  states, for any natural number  $n$ . (Recall that a safety property is one that, if violated, is violated at some definite point in the behavior.)

A more precise definition of  $\dashv$  appears in Section 16.2.4 (page 314). If  $E$  describes the desired behavior of the environment and  $M$  describes the desired behavior of the system, then we take as our open-system specification the formula  $E \dashv M$ .

Once we write separate specifications of the components, we can usually transform a complete-system specification into an open-system one by simply replacing conjunction with  $\dashv$ . This requires first deciding whether each conjunct of the complete-system specification belongs to the specification of the environment, of the system, or of neither. As an example, consider the composite specification of the FIFO buffer in module *CompositeFIFO* on page 143. We take the system to consist of just the buffer, with the sender and receiver forming the environment. The closed-system specification *Spec* has three main conjuncts:

*Sender*  $\wedge$  *Buffer*  $\wedge$  *Receiver*

The conjuncts *Sender* and *Receiver* are clearly part of the environment specification, and *Buffer* is part of the system specification.

$(in.ack = in.rdy) \wedge (out.ack = out.rdy)$

These two initial conjuncts can be assigned to either, depending on which component we want to blame if they are violated. Let's assign to the component sending on a channel  $c$  the responsibility for establishing that  $c.ack = c.rdy$  holds initially. We then assign  $in.ack = in.rdy$  to the environment and  $out.ack = out.rdy$  to the system.

$\square(IsChannel(in) \wedge IsChannel(out))$

This formula is not naturally attributed to either the system or the environment. We regard it as a property inherent in our way of modeling the system, which assumes that *in* and *out* are records with *ack*, *val*, and *rdy* components. We therefore take the formula to be a separate conjunct of the complete specification, not belonging to either the system or the environment.

We then have the following open-system specification for the FIFO buffer:

$$\begin{aligned} & \wedge \square(IsChannel(in) \wedge IsChannel(out)) \\ & \wedge (in.ack = in.rdy) \wedge \text{Sender} \wedge \text{Receiver} \dashv \\ & \quad (out.ack = out.rdy) \wedge \text{Buffer} \end{aligned}$$

As this example suggests, there is little difference between writing a composite complete-system specification and an open-system specification. Most of the

specification doesn't depend on which we choose. The two differ only at the very end, when we put the pieces together.

## 10.8 Interface Refinement

An *interface refinement* is a method of obtaining a lower-level specification by refining the variables of a higher-level specification. Let's start with two examples and then discuss interface refinement in general.

### 10.8.1 A Binary Hour Clock

In specifying an hour clock, we described its display with a variable *hr* whose value (in a behavior satisfying the specification) is an integer from 1 to 12. Suppose we want to specify a *binary hour clock*. This is an hour clock for use in a computer, where the display consists of a four-bit register that displays the hour as one of the twelve values 0001, 0010, . . . , 1100. We can easily specify such a clock from scratch. But suppose we want to describe it informally to someone who already knows what an hour clock is. We would simply say that a binary hour clock is the same as an ordinary hour clock, except that the value of the display is represented in binary. We now formalize that description.

We begin by describing what it means for a four-bit value to represent a number. There are several reasonable ways to represent a four-bit value mathematically. We could use a four-element sequence, which in TLA<sup>+</sup> is a function whose domain is 1 . . . 4. However, a mathematician would find it more natural to represent an  $(n+1)$ -bit number as a function from 0 . . .  $n$  to {0, 1}, the function  $b$  representing the number  $b[0] * 2^0 + b[1] * 2^1 + \dots + b[n] * 2^n$ . In TLA<sup>+</sup>, we can define *BitArrayVal*( $b$ ) to be the numerical value of such a function  $b$  by

$$\begin{aligned}
 \text{BitArrayVal}(b) &\triangleq \\
 \text{LET } n &\triangleq \text{CHOOSE } i \in \text{Nat} : \text{DOMAIN } b = 0 \dots i \\
 \text{val}[i \in 0 \dots n] &\triangleq \text{Defines } \text{val}[i] \text{ to equal } b[0] * 2^0 + \dots + b[i] * 2^i. \\
 &\text{IF } i = 0 \text{ THEN } b[0] * 2^0 \text{ ELSE } b[i] * 2^i + \text{val}[i - 1] \\
 &\text{IN } \text{val}[n]
 \end{aligned}$$

We can also write {0, 1} as 0 . . . 1.

To specify a binary hour clock whose display is described by the variable *bits*, we would simply say that *BitArrayVal*(*bits*) changes the same way that the specification *HC* of the hour clock allows *hr* to change. Mathematically, this means that we obtain the specification of the binary hour clock by substituting *BitArrayVal*(*bits*) for the variable *hr* in *HC*. In TLA<sup>+</sup>, substitution is expressed with the INSTANCE statement. Writing

$$B \triangleq \text{INSTANCE } \text{HourClock} \text{ WITH } \text{hr} \leftarrow \text{BitArrayVal}(\text{bits})$$

defines (among other things)  $B!HC$  to be the formula obtained from  $HC$  by substituting  $BitArrayVal(bits)$  for  $hr$ .

Unfortunately, this specification is wrong. The value of  $BitArrayVal(b)$  is specified only if  $b$  is a function with domain  $0 \dots n$  for some natural number  $n$ . We don't know what  $BitArrayVal(\{\text{``abc''}\})$  equals. It might equal 7. If it did, then  $B!HC$  would allow a behavior in which the initial value of  $bits$  is  $\{\text{``abc''}\}$ . We must rule out this possibility by substituting for  $hr$  not  $BitArrayVal(bits)$ , but some expression  $HourVal(bits)$  whose value is an element of  $1 \dots 12$  only if  $b$  is a function in  $[(0 \dots 3) \rightarrow \{0, 1\}]$ . For example, we can write

$$HourVal(b) \triangleq \begin{array}{ll} \text{IF } b \in [(0 \dots 3) \rightarrow \{0, 1\}] & \text{THEN } BitArrayVal(b) \\ & \text{ELSE } 99 \end{array}$$

$$B \triangleq \text{INSTANCE } HourClock \text{ WITH } hr \leftarrow HourVal(bits)$$

This defines  $B!HC$  to be the desired specification of the binary hour clock. Because  $HC$  is not satisfied by a behavior in which  $hr$  ever assumes the value 99,  $B!HC$  is not satisfied by any behavior in which  $bits$  ever assumes a value not in the set  $[(0 \dots 3) \rightarrow \{0, 1\}]$ .

There is another way to use the specification  $HC$  of the hour clock to specify the binary hour clock. Instead of substituting for  $hr$  in the hour-clock specification, we first specify a system consisting of both an hour clock and a binary hour clock that keep the same time, and we then hide the hour clock. This specification has the form

$$(10.7) \exists hr : IR \wedge HC$$

where  $IR$  is a temporal formula that is true iff  $bits$  is always the four-bit value representing the value of  $hr$ . This formula asserts that  $bits$  is the representation of  $hr$  as a four-bit array, for some choice of values for  $hr$  that satisfies  $HC$ . Using the definition of  $HourVal$  given above, we can define  $IR$  simply to equal  $\square(h = HourVal(b))$ .

If  $HC$  is defined as in module  $HourClock$ , then (10.7) can't appear in a  $TLA^+$  specification. For  $HC$  to be defined in the context of the formula, the variable  $hr$  must be declared in that context. If  $hr$  is already declared, then it can't be used as the bound variable of the quantifier  $\exists$ . As usual, this problem is solved with parametrized instantiation. The complete  $TLA^+$  specification  $BHC$  of the binary hour clock appears in module  $BinaryHourClock$  of Figure 10.3 on the next page.

## 10.8.2 Refining a Channel

As our second example of interface refinement, consider a system that interacts with its environment by sending numbers from 1 through 12 over a channel. We refine it to a lower-level system that is the same, except it sends a number

---

MODULE *BinaryHourClock*

---

EXTENDS *Naturals*

VARIABLE *bits*

$H(hr) \triangleq \text{INSTANCE } HourClock$

$\text{BitArrayVal}(b) \triangleq \text{LET } n \triangleq \text{CHOOSE } i \in Nat : \text{DOMAIN } b = 0 \dots i$

$\text{val}[i \in 0 \dots n] \triangleq \begin{array}{l} \text{Defines } \text{val}[i] \text{ to equal } b[0] * 2^0 + \dots + b[i] * 2^i. \\ \text{IF } i = 0 \text{ THEN } b[0] * 2^0 \text{ ELSE } (b[i] * 2^i) + \text{val}[i - 1] \end{array}$

IN  $\text{val}[n]$

$\text{HourVal}(b) \triangleq \text{IF } b \in [(0 \dots 3) \rightarrow \{0, 1\}] \text{ THEN } \text{BitArrayVal}(b)$

ELSE 99

$IR(b, h) \triangleq \square(h = \text{HourVal}(b))$

$BHC \triangleq \exists hr : IR(\text{bits}, hr) \wedge H(hr)!HC$

**Figure 10.3:** A specification of a binary hour clock.

as a sequence of four bits. Each bit is sent separately, starting with the left-most (most significant) one. For example, to send the number 5, the lower-level system sends the sequence of bits 0, 1, 0, 1. We specify both channels with the *Channel* module of Figure 3.2 on page 30, so each value that is sent must be acknowledged before the next one can be sent.

Suppose  $H\text{Spec}$  is the higher-level system's specification, and its channel is represented by the variable  $h$ . Let  $l$  be the variable representing the lower-level channel. We write the lower-level system's specification as

(10.8)  $\exists h : IR \wedge HSpec$

where  $IR$  specifies the sequence of values sent over  $h$  as a function of the values sent over  $l$ . The sending of the fourth bit on  $l$  is interpreted as the sending of the complete number on  $h$ ; the next acknowledgment on  $l$  is interpreted as the sending of the acknowledgment on  $h$ ; and any other step is interpreted as a step that doesn't change  $h$ .

To define *IR*, we instantiate module *Channel* for each of the channels:

$H \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow h, Data \leftarrow 1..12$

$L \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow l, Data \leftarrow \{0, 1\}$

*Data* is the set of values that can be sent over the channel.

Sending a value  $d$  over channel  $l$  is thus represented by an  $L!Send(d)$  step, and acknowledging receipt of a value on channel  $h$  is represented by an  $H!Rcv$  step. The following behavior represents sending and acknowledging a 5, where I have

omitted all steps that don't change  $l$ :

$$\begin{array}{ccccccc}
 s_0 & \xrightarrow{L!Send(0)} & s_1 & \xrightarrow{L!Rcv} & s_2 & \xrightarrow{L!Send(1)} & s_3 \\
 & & & & & \xrightarrow{L!Rcv} & s_4 \\
 & & s_5 & \xrightarrow{L!Rcv} & s_6 & \xrightarrow{L!Send(1)} & s_7 \\
 & & & & & \xrightarrow{L!Rcv} & s_8 \longrightarrow \dots
 \end{array}$$

This behavior will satisfy  $IR$  iff  $s_6 \rightarrow s_7$  is an  $H!Send(5)$  step,  $s_7 \rightarrow s_8$  is an  $H!Rcv$  step, and all the other steps leave  $h$  unchanged.

We want to make sure that (10.8) is not satisfied unless  $l$  represents a correct lower-level channel—for example, (10.8) should be false if  $l$  is set to some bizarre value. We will therefore define  $IR$  so that, if the sequence of values assumed by  $l$  does not represent a channel over which bits are sent and acknowledged, then the sequence of values of  $h$  does not represent a correct behavior of a channel over which numbers from 1 to 12 are sent. Formula  $HSpec$ , and hence (10.8), will then be false for such a behavior.

Formula  $IR$  will have the standard form for a TLA specification, with an initial condition and a next-state action. However, it specifies  $h$  as a function of  $l$ ; it does not constrain  $l$ . Therefore, the initial condition does not specify the initial value of  $l$ , and the next-state action does not specify the value of  $l'$ . (The value of  $l$  is constrained implicitly by  $IR$ , which asserts a relation between the values of  $h$  and  $l$ , together with the conjunct  $HSpec$  in (10.8), which constrains the value of  $h$ .) For the next-state action to specify the value sent on  $h$ , we need an internal variable that remembers what has been sent on  $l$  since the last complete number. We let the variable  $bitsSent$  contain the sequence of bits sent so far for the current number. For convenience,  $bitsSent$  contains the sequence of bits in reverse order, with the most recently-sent bit at the head. This means that the high-order bit of the number, which is sent first, is at the tail of  $bitsSent$ .

The definition of  $IR$  appears in module *ChannelRefinement* of Figure 10.4 on the next page. The module first defines *ErrorVal* to be an arbitrary value that is not a legal value of  $h$ . Next comes the definition of the function *BitSeqToNat*. If  $s$  is a sequence of bits, then  $BitSeqToNat[s]$  is its numeric value interpreted as a binary number whose low-order bit is at the head of  $s$ . For example  $BitSeqToNat[\langle 0, 1, 1 \rangle]$  equals 6. Then come the two instantiations of module *Channel*.

There follows a submodule that defines the internal specification—the one with the internal variable  $bitsSent$  visible. The internal specification's initial predicate *Init* asserts that if  $l$  has a legal initial value, then  $h$  can have any legal initial value; otherwise  $h$  has an illegal value. Initially  $bitsSent$  is the empty sequence  $\langle \rangle$ . The internal specification's next-state action is the disjunction of three actions:

*SendBit* A *SendBit* step is one in which a bit is sent on  $l$ . If  $bitsSent$  has fewer than three elements, so fewer than three bits have already been sent, then the bit is prepended to the head of  $bitsSent$  and  $h$

The use of a submodule to define an internal specification was introduced in the real-time hour-clock specification of Section 9.1.

MODULE *ChannelRefinement*

This module defines an interface refinement from a higher-level channel  $h$ , over which numbers in  $1 \dots 12$  are sent, to a lower-level channel  $l$  in which a number is sent as a sequence of four bits, each separately acknowledged. (See the *Channel* module in Figure 3.2 on page 30.) Formula *IR* is true iff the sequence of values assumed by  $h$  represents the higher-level view of the sequence of values sent on  $l$ . If the sequence of values assumed by  $l$  doesn't represent the sending and acknowledging of bits, then  $h$  assumes an illegal value.

EXTENDS *Naturals*, *Sequences*

VARIABLES  $h, l$

$ErrorVal \triangleq \text{CHOOSE } v : v \notin [val : 1 \dots 12, \text{rdy} : \{0, 1\}, \text{ack} : \{0, 1\}]$

$BitSeqToNat[s \in Seq(\{0, 1\})] \triangleq BitSeqToNat[\langle b_0, b_1, b_2, b_3 \rangle] = b_0 + 2 * (b_1 + 2 * (b_2 + 2 * b_3))$

IF  $s = \langle \rangle$  THEN 0 ELSE  $Head(s) + 2 * BitSeqToNat[Tail(s)]$

$H \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow h, Data \leftarrow 1 \dots 12$  H is a channel for sending numbers in  $1 \dots 12$ ;  $L$  is a channel for sending bits.

$L \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow l, Data \leftarrow \{0, 1\}$

MODULE *Inner*

VARIABLE  $bitsSent$  The sequence of the bits sent so far for the current number.

$Init \triangleq \wedge bitsSent = \langle \rangle$

$\wedge \text{IF } L!Init \text{ THEN } H!Init$  Defines the initial value of  $h$  as a function of  $l$ .

$\wedge \text{ELSE } h = ErrorVal$

$SendBit \triangleq \exists b \in \{0, 1\} :$

$\wedge L!Send(b)$

$\wedge \text{IF } Len(bitsSent) < 3$

$\text{THEN } \wedge bitsSent' = \langle b \rangle \circ bitsSent$

$\wedge \text{UNCHANGED } h$

$\text{ELSE } \wedge bitsSent' = \langle \rangle$

$\wedge H!Send(BitSeqToNat[\langle b \rangle \circ bitsSent])$

Sending one of the first three bits on  $l$  prepends it to the front of  $bitsSent$  and leaves  $h$  unchanged; sending the fourth bit resets  $bitsSent$  and sends the complete number on  $h$ .

$RcvBit \triangleq \wedge L!Rcv$

$\wedge \text{IF } bitsSent = \langle \rangle \text{ THEN } H!Rcv$

$\wedge \text{ELSE } \text{UNCHANGED } h$

$\wedge \text{UNCHANGED } bitsSent$

A  $Rcv$  action on  $l$  causes a  $Rcv$  action on  $h$  iff it follows the sending of the fourth bit.

$Error \triangleq \wedge l' \neq l$

$\wedge \neg((\exists b \in \{0, 1\} : L!Send(b)) \vee L!Rcv)$

$\wedge h' = ErrorVal$

An illegal action on  $l$  sets  $h$  to  $ErrorVal$ .

$Next \triangleq SendBit \vee RcvBit \vee Error$

$InnerIR \triangleq Init \wedge \square[Next]_{\langle l, h, bitsSent \rangle}$

$I(bitsSent) \triangleq \text{INSTANCE } Inner$

$IR \triangleq \exists bitsSent : I(bitsSent)!InnerIR$

Figure 10.4: Refining a channel.

is left unchanged. Otherwise, the value represented by the four bits sent so far, including the current bit, is sent on  $h$  and  $bitsSent$  is reset to  $\langle \rangle$ .

*RcvBit* A *RcvBit* step is one in which an acknowledgment is sent on  $l$ . It represents the sending of an acknowledgment on  $h$  iff this is an acknowledgment of the fourth bit, which is true iff  $bitsSent$  is the empty sequence.

*Error* An *Error* step is one in which an illegal change to  $l$  occurs. It sets  $h$  to an illegal value.

The inner specification *InnerIR* has the usual form. (There is no liveness requirement.) The outer module then instantiates the inner submodule with  $bitsSent$  as a parameter, and it defines *IR* to equal *InnerIR* with *bitsSent* hidden.

Now suppose we have a module *HigherSpec* that defines a specification *HSpec* of a system that communicates by sending numbers from 1 through 12 over a channel *hchan*. We obtain, as follows, a lower-level specification *LSpec* in which the numbers are sent as sequences of bits on a channel *lchan*. We first declare *lchan* and all the variables and constants of the *HigherSpec* module except *hchan*. We then write

$$\begin{aligned} HS(hchan) &\triangleq \text{INSTANCE } HigherSpec \\ CR(h) &\triangleq \text{INSTANCE } ChannelRefinement \text{ WITH } l \leftarrow lchan \\ LSpec &\triangleq \exists h : CR(h)!IR \wedge HS(h)!HSpec \end{aligned}$$

### 10.8.3 Interface Refinement in General

In the examples of the binary clock and of channel refinement, we defined a lower-level specification *LSpec* in terms of a higher-level one *HSpec* as

$$(10.9) \quad LSpec \triangleq \exists h : IR \wedge HSpec$$

where  $h$  is a free variable of *HSpec* and *IR* is a relation between  $h$  and the lower-level variable  $l$  of *LSpec*. We can view the internal specification *IR*  $\wedge$  *HSpec* as the composition of two components, as shown here:



We can regard *IR* as the specification of a component that transforms the lower-level behavior of  $l$  into the higher-level behavior of  $h$ . Formula *IR* is called an *interface refinement*.

In both examples, the interface refinement is independent of the system specification. It depends only on the representation of the interface—that is, on how the interaction between the system and its environment is represented. In general, for an interface refinement  $IR$  to be independent of the system using the interface, it should ascribe a behavior of the higher-level interface variable  $h$  to any behavior of the lower-level variable  $l$ . In other words, for any sequence of values for  $l$ , there should be some sequence of values for  $h$  that satisfy  $IR$ . This is expressed mathematically by the requirement that the formula  $\exists h : IR$  should be valid—that is, true for all behaviors.

So far, I have discussed refinement of a single interface variable  $h$  by a single variable  $l$ . This generalizes in the obvious way to the refinement of a collection of higher-level variables  $h_1, \dots, h_n$  by the variables  $l_1, \dots, l_m$ . The interface refinement  $IR$  specifies the values of the  $h_i$  in terms of the values of the  $l_j$  and perhaps of other variables as well. Formula (10.9) is replaced by

$$L\text{Spec} \triangleq \exists h_1, \dots, h_n : IR \wedge H\text{Spec}$$

A particularly simple type of interface refinement is a *data refinement*, in which  $IR$  has the form  $\square P$ , where  $P$  is a state predicate that expresses the values of the higher-level variables  $h_1, \dots, h_n$  as functions of the values of the lower-level variables  $l_1, \dots, l_m$ . The interface refinement in our binary clock specification is a data refinement, where  $P$  is the predicate  $hr = \text{HourVal}(\text{bits})$ . As another example, the two specifications of an asynchronous channel interface in Chapter 3 can each be obtained from the other by an interface refinement. The specification  $Spec$  of the *Channel* module (page 30) is equivalent to the specification obtained as a data refinement of the specification  $Spec$  of the *AsynchInterface* module (page 27) by letting  $P$  equal

$$(10.10) \quad chan = [val \mapsto val, rdy \mapsto rdy, ack \mapsto ack]$$

This formula asserts that  $chan$  is a record whose  $val$  field is the value of the variable  $val$ , whose  $rdy$  field is the value of the variable  $rdy$ , and whose  $ack$  field is the value of the variable  $ack$ . Conversely, specification  $Spec$  of the *AsynchInterface* module is equivalent to a data refinement of the specification  $Spec$  of the *Channel* module. In this case, defining the state predicate  $P$  is a little tricky. The obvious choice is to let  $P$  be the formula  $GoodVals$  defined by

$$\begin{aligned} GoodVals \triangleq & \wedge val = chan.val \\ & \wedge rdy = chan.rdy \\ & \wedge ack = chan.ack \end{aligned}$$

However, this can assert that  $val$ ,  $rdy$ , and  $ack$  have good values even if  $chan$  has an illegal value—for example, if it is a record with more than three fields. Instead, we let  $P$  equal

$$\begin{aligned} \text{IF } chan \in [val : Data, rdy : \{0, 1\}, ack : \{0, 1\}] \text{ THEN } GoodVals \\ \text{ELSE } BadVals \end{aligned}$$

where *BadVals* asserts that *val*, *rdy*, and *ack* have some illegal values—that is, values that are impossible in a behavior satisfying formula *Spec* of module *AsynchInterface*. (We don’t need such a trick when defining *chan* as a function of *val*, *rdy*, and *ack* because (10.10) implies that the value of *chan* is legal iff the values of all three variables *val*, *rdy*, and *ack* are legal.)

Data refinement is the simplest form of interface refinement. In a more complicated interface refinement, the value of the higher-level variables cannot be expressed as a function of the current values of the lower-level variables. In the channel refinement example of Section 10.8.2, the number being sent on the higher-level channel depends on the values of bits that were previously sent on the lower-level channel, not just on the lower-level channel’s current state.

We often refine both a system and its interface at the same time. For example, we may implement a specification *H* of a system that communicates by sending numbers over a channel with a lower-level specification *LImpl* of a system that sends individual bits. In this case, *LImpl* is not itself obtained from *HSpec* by an interface refinement. Rather, *LImpl* implements some specification *LSpec* that is obtained from *HSpec* by an interface refinement *IR*. In that case, we say that *LImpl* implements *HSpec* under the interface refinement *IR*.

### 10.8.4 Open-System Specifications

So far, we have considered interface refinement for complete-system specifications. Let’s now consider what happens if the higher-level specification *HSpec* is the kind of open-system specification discussed in Section 10.7 above. For simplicity, we consider the refinement of a single higher-level interface variable *h* by a single lower-level variable *l*. The generalization to more variables will be obvious.

Let’s suppose first that *HSpec* is a safety property, with no liveness condition. As explained in Section 10.7, the specification attributes each change to *h* either to the system or to the environment. Any change to a lower-level interface variable *l* that produces a change to *h* is therefore attributed to the system or the environment. A bad change to *h* that is attributed to the environment makes *HSpec* true; a bad change that is attributed to the system makes *HSpec* false. Thus, (10.9) defines *LSpec* to be an open-system specification. For this to be a sensible specification, the interface refinement *IR* must ensure that the way changes to *l* are attributed to the system or environment is sensible.

If *HSpec* contains liveness conditions, then interface refinement can be more subtle. Suppose *IR* is the interface refinement defined in the *ChannelRefinement* module of Figure 10.4 on page 162, and suppose that *HSpec* requires that the system eventually send some number on *h*. Consider a behavior in which the system sends the first bit of a number on *l*, but the environment never acknowledges it. Under the interface refinement *IR*, this behavior is interpreted as one in which *h* never changes. Such a behavior fails to satisfy the liveness condition

of  $HSpec$ . Thus, if  $LSpec$  is defined by (10.9), then failure of the environment to do something can cause  $LSpec$  to be violated, through no fault of the system.

In this example, we want the environment to be at fault if it causes the system to halt by failing to acknowledge any of the first three bits of a number sent by the system. (The acknowledgment of the fourth bit is interpreted by  $IR$  as the acknowledgment of a value sent on  $h$ , so blame for its absence is properly assigned to the environment.) Putting the environment at fault means making  $LSpec$  true. We can achieve this by modifying (10.9) to define  $LSpec$  as follows:

$$(10.11) \quad LSpec \triangleq Liveness \Rightarrow \exists h : IR \wedge HSpec$$

where *Liveness* is a formula requiring that any bit sent on  $l$ , other than the last bit of a number, must eventually be acknowledged. However, if  $l$  is set to an illegal value, then we want the safety part of the specification to determine who is responsible. So, we want *Liveness* to be true in this case.

We define *Liveness* in terms of the inner variables  $h$  and  $bitsSent$ , which are related to  $l$  by formula *InnerIR* from the *Inner* submodule of module *ChannelRefinement*. (Remember that  $l$  should be the only free variable of  $LSpec$ .) The action that acknowledges receipt of one of the first three bits of the number is  $RcvBit \wedge (bitsSent \neq \langle \rangle)$ . Weak fairness of this action asserts that the required acknowledgments must eventually be sent. For the case of illegal values, recall that sending a bad value on  $l$  causes  $h$  to equal *ErrorVal*. We want *Liveness* to be true if this ever happens, which means if it eventually happens. We therefore add the following definition to the submodule *Inner* of the *ChannelRefinement* module:

$$\begin{aligned} InnerLiveness &\triangleq \wedge \text{InnerIR} \\ &\wedge \vee \text{WF}_{\langle l, h, bitsSent \rangle}(RcvBit \wedge (bitsSent \neq \langle \rangle)) \\ &\vee \diamond(h = ErrorVal) \end{aligned}$$

To define *Liveness*, we have to hide  $h$  and  $bitsSent$  in *InnerLiveness*. We can do this, in a context in which  $l$  is declared, as follows:

$$\begin{aligned} ICR(h) &\triangleq \text{INSTANCE ChannelRefinement} \\ Liveness &\triangleq \exists h, bitsSent : ICR(h)!I(bitsSent)!InnerLiveness \end{aligned}$$

Now, suppose it is the environment that sends numbers over  $h$  and the system is supposed to acknowledge their receipt and then process them in some way. In this case, we want failure to acknowledge a bit to be a system error. So,  $LSpec$  should be false if *Liveness* is. The specification should then be

$$LSpec \triangleq Liveness \wedge (\exists h : IR \wedge HSpec)$$

Since  $h$  does not occur free in *Liveness*, this definition is equivalent to

$$LSpec \triangleq \exists h : Liveness \wedge IR \wedge HSpec$$

which has the form (10.9) if the interface refinement  $IR$  of (10.9) is taken to be  $Liveness \wedge IR$ . In other words, we can make the liveness condition part of the interface refinement. (In this case, we can simplify the definition by adding liveness directly to  $InnerIR$ .)

In general, if  $H\text{Spec}$  is an open-system specification that describes liveness as well as safety, then an interface refinement may have to take the form of formula (10.11). Both  $Liveness$  and the liveness condition of  $IR$  may depend on which changes to the lower-level interface variable  $l$  are attributed to the system and which to the environment. For the channel refinement, this means that they will depend on whether the system or the environment is sending values on the channel.

## 10.9 Should You Compose?

When specifying a system, should we write a monolithic specification with a single next-state action, a closed-system composition that is the conjunction of specifications of individual components, or an open-system specification? The answer is: it usually makes little difference. For a real system, the definitions of the components' actions will take hundreds or thousands of lines. The different forms of specification differ only in the few lines where we assemble the initial predicates and next-state actions into the final formula.

If you are writing a specification from scratch, it's probably better to write a monolithic specification. It is usually easier to understand. Of course, there are exceptions. We write a real-time specification as the conjunction of an untimed specification and timing constraints; describing the changes to the system variables and the timers with a single next-state action usually makes the specification harder to understand.

Writing a composite specification may be sensible when you are starting from an existing specification. If you already have a specification of one component, you may want to write a separate specification of the other component and compose the two specifications. If you have a higher-level specification, you may want to write a lower-level version as an interface refinement. However, these are rather rare situations. Moreover, it's likely to be just as easy to modify the original specification or reuse it in another way. For example, instead of conjoining a new component to the specification of an existing one, you can simply include the definition of the existing component's next-state action, with an EXTENDS or INSTANCE statement, as part of the new specification.

Composition provides a new way of writing a complete-system specification; it doesn't change the specification. The choice between a composite specification and a monolithic one is therefore ultimately a matter of taste. Disjoint-state compositions are generally straightforward and present no problems. Shared-state compositions can be tricky and require care.

Open-system specifications introduce a mathematically different kind of specification. A closed-system specification  $E \wedge M$  and its open-system counterpart  $E \dashv\rightarrow M$  are not equivalent. If we really want a specification to serve as a legal contract between a user and an implementer, then we have to write an open-system specification. We also need open-system specifications if we want to specify and reason about systems built by composing off-the-shelf components with pre-existing specifications. All we can assume about such a component is that it satisfies a contract between the system builder and the supplier, and such a contract can be formalized only as an open-system specification. However, you are unlikely to encounter off-the-shelf component specifications during the early part of the twenty-first century. In the near future, open-system specifications are likely to be of theoretical interest only.

# Chapter 11

## Advanced Examples

It would be nice to provide an assortment of typical examples that cover most of the specification problems that arise in practice. However, there is no such thing as a typical specification. Every real specification seems to pose its own problems. But we can partition all specifications into two classes, depending on whether or not they contain `VARIABLE` declarations.

A specification with no variables defines data structures and operations on those structures. For example, the *Sequences* module defines various operations on sequences. When specifying a system, you may need some kind of data structure other than the ones provided by the standard modules like *Sequences* and *Bags*, described in Chapter 18. Section 11.1 gives some examples of data structure specifications.

A system specification contains variables that represent the system's state. We can further divide system specifications into two classes—high-level specifications that describe what it means for a system to be correct, and lower-level specifications that describe what the system actually does. In the memory example of Chapter 5, the linearizable memory specification of Section 5.3 is a high-level specification of correctness, while the write-through cache specification of Section 5.6 describes how a particular algorithm works. This distinction is not precise; whether a specification is high- or low-level is a matter of perspective. But it can be a useful way of categorizing system specifications.

Lower-level system specifications tend to be relatively straightforward. Once the level of abstraction has been chosen, writing the specification is usually just a matter of getting the details right when describing what the system does. Specifying high-level correctness can be much more subtle. Section 11.2 considers a high-level specification problem—formally specifying a multiprocessor memory.

## 11.1 Specifying Data Structures

Most of the data structures required for writing specifications are mathematically simple and are easy to define in terms of sets, functions, and records. Section 11.1.2 describes the specification of one such structure—a graph. On rare occasions, a specification will require sophisticated mathematical concepts. The only examples I know of are hybrid system specifications, discussed in Section 9.5. There, we used a module for describing the solutions to differential equations. That module is specified in Section 11.1.3 below. Section 11.1.4 considers the tricky problem of defining operators for specifying BNF grammars. Although not the kind of data structure you’re likely to need for a system specification, specifying BNF grammars provides a nice little exercise in “mathematization”. The module developed in that section is used in Chapter 15 for specifying the grammar of TLA<sup>+</sup>. But, before specifying data structures, you should know how to make local definitions.

### 11.1.1 Local Definitions

In the course of specifying a system, we write lots of auxiliary definitions. A system specification may consist of a single formula *Spec*, but we define dozens of other identifiers in terms of which we define *Spec*. These other identifiers often have fairly common names—for example, the identifier *Next* is defined in many specifications. The different definitions of *Next* don’t conflict with one another because, if a module that defines *Next* is used as part of another specification, it is usually instantiated with renaming. For example, the *Channel* module is used in module *InnerFIFO* on page 38 with the statement

*InChan*  $\triangleq$  INSTANCE *Channel* WITH ...

The action *Next* of the *Channel* module is then instantiated as *InChan!**Next*, so its definition doesn’t conflict with the definition of *Next* in the *InnerFIFO* module.

A module that defines operations on a data structure is likely to be used in an EXTENDS statement, which does no renaming. The module might define some auxiliary operators that are used only to define the operators in which we’re interested. For example, we need the *DifferentialEquations* module only to define the single operator *Integrate*. However, *Integrate* is defined in terms of other defined operators with names like *Nbhd* and *IsDeriv*. We don’t want these definitions to conflict with other uses of those identifiers in a module that extends *DifferentialEquations*. So, we want the definitions of *Nbhd* and *IsDeriv* to be local to the *DifferentialEquations* module.<sup>1</sup>

---

<sup>1</sup>We could use the LET construct to put these auxiliary definitions inside the definition of *Integrate*, but that trick wouldn’t work if the *DifferentialEquations* module exported other operators besides *Integrate* that were defined in terms of *Nbhd* and *IsDeriv*.

TLA<sup>+</sup> provides a LOCAL modifier for making definitions local to a module. If a module  $M$  contains the definition

LOCAL  $\text{Foo}(x) \triangleq \dots$

then  $\text{Foo}$  can be used inside module  $M$  just like any ordinary defined identifier. However, a module that extends or instantiates  $M$  does not obtain the definition of  $\text{Foo}$ . That is, the statement EXTENDS  $M$  in another module does not define  $\text{Foo}$  in that module. Similarly, the statement

$N \triangleq \text{INSTANCE } M$

does not define  $N!\text{Foo}$ . The LOCAL modifier can also be applied to an instantiation. The statement

LOCAL INSTANCE  $\text{Sequences}$

in module  $M$  incorporates into  $M$  the definitions from the  $\text{Sequences}$  module. However, another module that extends or instantiates  $M$  does not obtain those definitions. Similarly, a statement like

LOCAL  $P(x) \triangleq \text{INSTANCE } N$

makes all the instantiated definitions local to the current module.

The LOCAL modifier can be applied only to definitions and INSTANCE statements. It cannot be applied to a declaration or to an EXTENDS statement, so you *cannot* write either of the following:

LOCAL CONSTANT  $N$

These are not legal statements.

LOCAL EXTENDS  $\text{Sequences}$

If a module has no CONSTANT or VARIABLE declarations and no submodules, then extending it and instantiating it are equivalent. Thus, the two statements

EXTENDS  $\text{Sequences}$       INSTANCE  $\text{Sequences}$

are equivalent.

In a module that defines general mathematical operators, I like to make all definitions local except for the ones that users of the module would expect. For example, users expect the  $\text{Sequences}$  module to define operators on sequences, such as  $\text{Append}$ . They don't expect it to define operators on numbers, such as  $+$ . The  $\text{Sequences}$  module uses  $+$  and other operators defined in the  $\text{Naturals}$  module. But instead of extending  $\text{Naturals}$ , it defines those operators with the statement

LOCAL INSTANCE  $\text{Naturals}$

The definitions of the operators from  $\text{Naturals}$  are therefore local to  $\text{Sequences}$ . A module that extends the  $\text{Sequences}$  module could then define  $+$  to mean something other than addition of numbers.

## 11.1.2 Graphs

A graph is an example of the kind of simple data structure often used in specifications. Let's now write a *Graphs* module for use in writing system specifications.

We must first decide how to represent a graph in terms of data structures that are already defined—either built-in TLA<sup>+</sup> data structures like functions, or ones defined in existing modules. Our decision depends on what kind of graphs we want to represent. Are we interested in directed graphs or undirected graphs? Finite or infinite graphs? Graphs with or without self-loops (edges from a node to itself)? If we are specifying graphs for a particular specification, the specification will tell us how to answer these questions. In the absence of such guidance, let's handle arbitrary graphs. My favorite way of representing both directed and undirected graphs is to specify arbitrary directed graphs, and to define an undirected graph as a directed graph that contains an edge iff it contains the opposite-pointing edge. Directed graphs have a pretty obvious representation: a directed graph consists of a set of nodes and a set of edges, where an edge from node  $m$  to node  $n$  is represented by the ordered pair  $\langle m, n \rangle$ .

In addition to deciding how to represent graphs, we must decide how to structure the *Graphs* module. The decision depends on how we expect the module to be used. For a specification that uses a single graph, it is most convenient to define operations on that specific graph. So, we want the *Graphs* module to have (constant) parameters *Node* and *Edge* that represent the sets of nodes and edges of a particular graph. A specification could use such a module with a statement

```
INSTANCE Graphs WITH Node ← ... , Edge ← ...
```

where the “...”s are the sets of nodes and edges of the particular graph appearing in the specification. On the other hand, a specification might use many different graphs. For example, it might include a formula that asserts the existence of a subgraph, satisfying certain properties, of some given graph  $G$ . Such a specification needs operators that take a graph as an argument—for example, a *Subgraph* operator defined so  $\text{Subgraph}(G)$  is the set of all subgraphs of a graph  $G$ . In this case, the *Graphs* module would have no parameters, and specifications would incorporate it with an EXTENDS statement. Let's write this kind of module.

An operator like *Subgraph* takes a graph as an argument, so we have to decide how to represent a graph as a single value. A graph  $G$  consists of a set  $N$  of nodes and a set  $E$  of edges. A mathematician would represent  $G$  as the ordered pair  $\langle N, E \rangle$ . However,  $G.\text{node}$  is more perspicuous than  $G[1]$ , so we represent  $G$  as a record with *node* field  $N$  and *edge* field  $E$ .

Having made these decisions, it's easy to define any standard operator on graphs. We just have to decide what we should define. Here are some generally useful operators:

*IsDirectedGraph*( $G$ )

True iff  $G$  is an arbitrary directed graph—that is, a record with *node* field  $N$  and *edge* field  $E$  such that  $E$  is a subset of  $N \times N$ . This operator is useful because a specification might want to assert that something is a directed graph. (To understand how to assert that  $G$  is a record with *node* and *edge* fields, see the definition of *IsChannel* in Section 10.3 on page 140.)

*DirectedSubgraph*( $G$ )

The set of all subgraphs of a directed graph  $G$ . Alternatively, we could define *IsDirectedSubgraph*( $H, G$ ) to be true iff  $H$  is a subgraph of  $G$ . However, it's easy to express *IsDirectedSubgraph* in terms of *DirectedSubgraph*:

$$\text{IsDirectedSubgraph}(H, G) \equiv H \in \text{DirectedSubgraph}(G)$$

On the other hand, it's awkward to express *DirectedSubgraph* in terms of *IsDirectedSubgraph*:

$$\begin{aligned} \text{DirectedSubgraph}(G) &= \\ \text{CHOOSE } S : \forall H : (H \in S) &\equiv \text{IsDirectedSubgraph}(H, G) \end{aligned}$$

Section 6.1 explains why we can't define a set of all directed graphs, so we had to define the *IsDirectedGraph* operator.

*IsUndirectedGraph*( $G$ )*UndirectedSubgraph*( $G$ )

These are analogous to the operators for directed graphs. As mentioned above, an undirected graph is a directed graph  $G$  such that for every edge  $\langle m, n \rangle$  in  $G.\text{edge}$ , the inverse edge  $\langle n, m \rangle$  is also in  $G.\text{edge}$ . Note that *DirectedSubgraph*( $G$ ) contains directed graphs that are not undirected graphs—except for certain “degenerate” graphs  $G$ , such as graphs with no edges.

*Path*( $G$ )

The set of all paths in  $G$ , where a path is any sequence of nodes that can be obtained by following edges in the direction they point. This definition is useful because many properties of a graph can be expressed in terms of its set of paths. It is convenient to consider the one-element sequence  $\langle n \rangle$  to be a path, for any node  $n$ .

*AreConnectedIn*( $m, n, G$ )

True iff there is a path from node  $m$  to node  $n$  in  $G$ . The utility of this operator becomes evident when you try defining various common graph properties, like connectivity.

There are any number of other graph properties and classes of graphs that we might define. Let's define these two:

*IsStronglyConnected*( $G$ )

True iff  $G$  is strongly connected, meaning that there is a path from any node to any other node. For an undirected graph, strongly connected is equivalent to the ordinary definition of connected.

*IsTreeWithRoot*( $G, r$ )

True iff  $G$  is a tree with root  $r$ , where we represent a tree as a graph with an edge from each nonroot node to its parent. Thus, the parent of a nonroot node  $n$  equals

CHOOSE  $m \in G.\text{node} : \langle n, m \rangle \in G.\text{edge}$

The *Graphs* module appears on the next page. By now, you should be able to work out for yourself the meanings of all the definitions.

### 11.1.3 Solving Differential Equations

Section 9.5 on page 132 describes how to specify a hybrid system whose state includes a physical variable satisfying an ordinary differential equation. The specification uses an operator *Integrate* such that  $\text{Integrate}(D, t_0, t_1, \langle x_0, \dots, x_{n-1} \rangle)$  is the value at time  $t_1$  of the  $n$ -tuple

$$\langle x, dx/dt, \dots, d^{n-1}x/dt^{n-1} \rangle$$

where  $x$  is a solution to the differential equation

$$D[t, x, dx/dt, \dots, d^n x/dt^n] = 0$$

whose 0<sup>th</sup> through  $(n-1)^{\text{st}}$  derivatives at time  $t_0$  are  $x_0, \dots, x_{n-1}$ . We assume that there is such a solution and that it is unique. Defining *Integrate* illustrates how to express sophisticated mathematics in TLA<sup>+</sup>.

We start by defining some mathematical notation that we will use to define the derivative. As usual, we obtain from the *Reals* module the definitions of the set *Real* of real numbers and of the ordinary arithmetic operators. Let *PosReal* be the set of all positive reals:

$$\text{PosReal} \triangleq \{r \in \text{Real} : r > 0\}$$

and let *OpenInterval*( $a, b$ ) be the open interval from  $a$  to  $b$  (the set of numbers greater than  $a$  and less than  $b$ ):

$$\text{OpenInterval}(a, b) \triangleq \{s \in \text{Real} : (a < s) \wedge (s < b)\}$$

(Mathematicians usually write this set as  $(a, b)$ .) Let's also define *Nbhd*( $r, e$ ) to be the open interval of width  $2e$  centered at  $r$ :

$$\text{Nbhd}(r, e) \triangleq \text{OpenInterval}(r - e, r + e)$$

MODULE *Graphs*

A module that defines operators on graphs. A directed graph is represented as a record whose *node* field is the set of nodes and whose *edge* field is the set of edges, where an edge is an ordered pair of nodes.

LOCAL INSTANCE *Naturals*

LOCAL INSTANCE *Sequences*

$IsDirectedGraph(G) \triangleq$  True iff  $G$  is a directed graph.

$\wedge G = [node \mapsto G.\text{node}, \text{edge} \mapsto G.\text{edge}]$

$\wedge G.\text{edge} \subseteq (G.\text{node} \times G.\text{node})$

$DirectedSubgraph(G) \triangleq$  The set of all (directed) subgraphs of a directed graph.

$\{H \in [node : \text{SUBSET } G.\text{node}, \text{edge} : \text{SUBSET } (G.\text{node} \times G.\text{node})] :$

$IsDirectedGraph(H) \wedge H.\text{edge} \subseteq G.\text{edge}\}$

$IsUndirectedGraph(G) \triangleq$  An undirected graph is a directed graph in which every

$\wedge IsDirectedGraph(G)$  edge has an oppositely directed one.

$\wedge \forall e \in G.\text{edge} : \langle e[2], e[1] \rangle \in G.\text{edge}$

$UndirectedSubgraph(G) \triangleq$  The set of (undirected) subgraphs of an undirected graph.

$\{H \in DirectedSubgraph(G) : IsUndirectedGraph(H)\}$

$Path(G) \triangleq$  The set of paths in  $G$ , where a path is represented as a sequence of nodes.

$\{p \in Seq(G.\text{node}) : \wedge p \neq \langle \rangle$

$\wedge \forall i \in 1 \dots (\text{Len}(p) - 1) : \langle p[i], p[i + 1] \rangle \in G.\text{edge}\}$

$AreConnectedIn(m, n, G) \triangleq$  True iff there is a path from  $m$  to  $n$  in graph  $G$ .

$\exists p \in Path(G) : (p[1] = m) \wedge (p[\text{Len}(p)] = n)$

$IsStronglyConnected(G) \triangleq$  True iff graph  $G$  is strongly connected.

$\forall m, n \in G.\text{node} : AreConnectedIn(m, n, G)$

$IsTreeWithRoot(G, r) \triangleq$  True if  $G$  is a tree with root  $r$ , where edges point

$\wedge IsDirectedGraph(G)$  from child to parent.

$\wedge \forall e \in G.\text{edge} : \wedge e[1] \neq$

$\wedge \forall f \in G.\text{edge} : (e[1] = f[1]) \Rightarrow (e = f)$

$\wedge \forall n \in G.\text{node} : AreConnectedIn(n, r, G)$

Figure 11.1: A module for specifying operators on graphs.

To explain the definitions, we need some notation for the derivative of a function. It's rather difficult to make mathematical sense of the usual notation  $df/dt$  for the derivative of  $f$ . (What exactly is  $t$ ?) So, let's use a mathematically simpler notation and write the  $n^{\text{th}}$  derivative of the function  $f$  as  $f^{(n)}$ . (We don't have to use TLA<sup>+</sup> notation because differentiation will not appear explicitly in our definitions.) Recall that  $f^{(0)}$ , the 0<sup>th</sup> derivative of  $f$ , equals  $f$ .

We can now start to define *Integrate*. If  $a$  and  $b$  are numbers, *InitVals* is an  $n$ -tuple of numbers, and  $D$  is a function from  $(n + 2)$ -tuples of numbers to numbers, then

$$\text{Integrate}(D, a, b, \text{InitVals}) = \langle f^{(0)}[b], \dots, f^{(n-1)}[b] \rangle$$

where  $f$  is the function satisfying the following two conditions:

- $D[r, f^{(0)}[r], f^{(1)}[r], \dots, f^{(n)}[r]] = 0$ , for all  $r$  in some open interval containing  $a$  and  $b$ .
- $\langle f^{(0)}[a], \dots, f^{(n-1)}[a] \rangle = \text{InitVals}$

We want to define  $\text{Integrate}(D, a, b, \text{InitVals})$  in terms of this function  $f$ , which we can specify using the CHOOSE operator. It's easiest to choose not just  $f$ , but its first  $n$  derivatives as well. So, we choose a function  $g$  such that  $g[i] = f^{(i)}$  for  $i \in 0 \dots n$ . The function  $g$  maps numbers in  $0 \dots n$  into functions. More precisely,  $g$  is an element of

$$[0 \dots n \rightarrow [\text{OpenInterval}(a - e, b + e) \rightarrow \text{Real}]]$$

for some positive  $e$ . It is the function in this set that satisfies the following conditions:

1.  $g[i]$  is the  $i^{\text{th}}$  derivative of  $g[0]$ , for all  $i \in 0 \dots n$ .
2.  $D[r, g[0][r], \dots, g[n][r]] = 0$ , for all  $r$  in  $\text{OpenInterval}(a - e, b + e)$ .
3.  $\langle g[0][a], \dots, g[n-1][a] \rangle = \text{InitVals}$

We now have to express these conditions formally.

To express the first condition, we will define *IsDeriv* so that  $\text{IsDeriv}(i, df, f)$  is true iff  $df$  is the  $i^{\text{th}}$  derivative of  $f$ . More precisely, this will be the case if  $f$  is a real-valued function on an open interval; we don't care what  $\text{IsDeriv}(i, df, f)$  equals for other values of  $f$ . Condition 1 is then

$$\forall i \in 1 \dots n : \text{IsDeriv}(i, g[i], g[0])$$

To express the second condition formally, without the “ $\dots$ ”, we reason as follows:

$$\begin{aligned} D[r, g[0][r], \dots, g[n][r]] \\ = D[\langle r, g[0][r], \dots, g[n][r] \rangle] & \quad \text{See page 50.} \\ = D[\langle r \rangle \circ \langle g[0][r], \dots, g[n][r] \rangle] & \quad \text{Tuples are sequences} \\ = D[\langle r \rangle \circ [i \in 1 \dots (n+1) \mapsto g[i-1][r]]] & \quad \text{An } (n+1)\text{-tuple is a function with domain } 1 \dots n+1. \end{aligned}$$

The third condition is simply

$$\forall i \in 1 \dots n : g[i-1][a] = \text{InitVals}[i]$$

We can therefore write the formula specifying  $g$  as

$$\begin{aligned} \exists e \in \text{PosReal} : \wedge g \in [0 \dots n \rightarrow [\text{OpenInterval}(a - e, b + e) \rightarrow \text{Real}]] \\ \wedge \forall i \in 1 \dots n : \wedge \text{IsDeriv}(i, g[i], g[0]) \\ \wedge g[i-1][a] = \text{InitVals}[i] \\ \wedge \forall r \in \text{OpenInterval}(a - e, b + e) : \\ D[\langle r \rangle \circ [i \in 1 \dots (n+1) \mapsto g[i-1][r]]] = 0 \end{aligned}$$

where  $n$  is the length of  $\text{InitVals}$ . The value of  $\text{Integrate}(D, a, b, \text{InitVals})$  is the tuple  $\langle g[0][b], \dots, g[n-1][b] \rangle$ , which can be written formally as

$$[i \in 1 \dots n \mapsto g[i-1][b]]$$

To complete the definition of  $\text{Integrate}$ , we now define the operator  $\text{IsDeriv}$ . It's easy to define the  $i^{\text{th}}$  derivative inductively in terms of the first derivative. So, we define  $\text{IsFirstDeriv}(df, f)$  to be true iff  $df$  is the first derivative of  $f$ , assuming that  $f$  is a real-valued function whose domain is an open interval. Our definition actually works if the domain of  $f$  is any open set.<sup>2</sup> Elementary calculus tells us that  $df[r]$  is the derivative of  $f$  at  $r$  iff

$$df[r] = \lim_{s \rightarrow r} \frac{f[s] - f[r]}{s - r}$$

The classical “ $\delta$ - $\epsilon$ ” definition of the limit states that this is true iff, for every  $\epsilon > 0$ , there is a  $\delta > 0$  such that  $0 < |s - r| < \delta$  implies

$$\left| df[r] - \frac{f[s] - f[r]}{s - r} \right| < \epsilon$$

Stated formally, this condition is

$$\begin{aligned} \forall \epsilon \in \text{PosReal} : \\ \exists \delta \in \text{PosReal} : \\ \forall s \in \text{Nbhd}(r, \delta) \setminus \{r\} : \frac{f[s] - f[r]}{s - r} \in \text{Nbhd}(df[r], \epsilon) \end{aligned}$$

We define  $\text{IsFirstDeriv}(df, f)$  to be true iff the domains of  $df$  and  $f$  are equal, and this condition holds for all  $r$  in their domain.

The definitions of  $\text{Integrate}$  and all the other operators introduced above appear in the *DifferentialEquations* module of Figure 11.2 on the next page. The LOCAL construct described in Section 11.1.1 above is used to make all these definitions local to the module, except for the definition of  $\text{Integrate}$ .

<sup>2</sup>A set  $S$  is open iff for every  $r \in S$  there exists an  $\epsilon > 0$  such that the interval from  $r - \epsilon$  to  $r + \epsilon$  is contained in  $S$ .

MODULE *DifferentialEquations*

This module defines the operator *Integrate* for specifying the solution to a differential equation. If  $a$  and  $b$  are reals with  $a \leq b$ ; *InitVals* is an  $n$ -tuple of reals; and  $D$  is a function from  $(n+1)$ -tuples of reals to reals; then this is the  $n$ -tuple of values

$$\langle f[b], \frac{df}{dt}[b], \dots, \frac{d^{n-1}f}{dt^{n-1}}[b] \rangle$$

where  $f$  is the solution to the differential equation

$$D[t, f, \frac{df}{dt}, \dots, \frac{d^n f}{dt^n}] = 0$$

such that

$$\langle f[a], \frac{df}{dt}[a], \dots, \frac{d^{n-1}f}{dt^{n-1}}[a] \rangle = \text{InitVals}$$

LOCAL INSTANCE *Reals*

LOCAL INSTANCE *Sequences*

LOCAL *PosReal*  $\triangleq \{r \in \text{Real} : r > 0\}$

LOCAL *OpenInterval*( $a, b$ )  $\triangleq \{s \in \text{Real} : (a < s) \wedge (s < b)\}$

LOCAL *Nbhd*( $r, e$ )  $\triangleq \text{OpenInterval}(r - e, r + e)$

The INSTANCE statement and these definitions are local, so a module that extends this one obtains only the definition of *Integrate*.

LOCAL *IsFirstDeriv*( $df, f$ )  $\triangleq$

$\wedge df \in [\text{DOMAIN } f \rightarrow \text{Real}]$

$\wedge \forall r \in \text{DOMAIN } f :$

$\forall e \in \text{PosReal} :$

$\exists d \in \text{PosReal} :$

$\forall s \in \text{Nbhd}(r, d) \setminus \{r\} : (f[s] - f[r])/(s - r) \in \text{Nbhd}(df[r], e)$

Assuming  $\text{DOMAIN } f$  is an open subset of *Real*, this is true iff  $f$  is differentiable and  $df$  is its first derivative. Recall that the derivative of  $f$  at  $r$  is the number  $df[r]$  satisfying the following condition: for every  $\epsilon$  there exists a  $\delta$  such that  $0 < |s - r| < \delta$  implies  $|df[r] - (f[s] - f[r])/(s - r)| < \epsilon$ .

LOCAL *IsDeriv*( $n, df, f$ )  $\triangleq$  True iff  $f$  is  $n$  times differentiable and  $df$  is its  $n^{\text{th}}$  derivative.

LET *IsD*[ $k \in 0 \dots n, g \in [\text{DOMAIN } f \rightarrow \text{Real}]$ ]  $\triangleq$   $\text{IsD}[k, g] = \text{IsDeriv}(k, g, f)$

IF  $k = 0$  THEN  $g = f$

ELSE  $\exists gg \in [\text{DOMAIN } f \rightarrow \text{Real}] : \wedge \text{IsFirstDeriv}(g, gg)$

$\wedge \text{IsD}[k - 1, gg]$

IN *IsD*[ $n, df$ ]

*Integrate*( $D, a, b, \text{InitVals}$ )  $\triangleq$

LET  $n \triangleq \text{Len}(\text{InitVals})$

$gg \triangleq \text{CHOOSE } g : \exists e \in \text{PosReal} : \wedge g \in [0 \dots n \rightarrow [\text{OpenInterval}(a - e, b + e) \rightarrow \text{Real}]]$

$\wedge \forall i \in 1 \dots n : \wedge \text{IsDeriv}(i, g[i], g[0])$

$\wedge g[i - 1][a] = \text{InitVals}[i]$

$\wedge \forall r \in \text{OpenInterval}(a - e, b + e) :$

$D[\langle r \rangle \circ [i \in 1 \dots (n + 1) \mapsto g[i - 1][r]]] = 0$

IN  $[i \in 1 \dots n \mapsto gg[i - 1][b]]$

**Figure 11.2:** A module for specifying the solution to a differential equation.

### 11.1.4 BNF Grammars

BNF, which stands for Backus-Naur Form, is a standard way of describing the syntax of computer languages. This section develops the *BNFGrammars* module, which defines operators for writing BNF grammars. A BNF grammar isn't the kind of data structure that arises in system specification, and  $TLA^+$  is not particularly well suited to specifying one. Its syntax doesn't allow us to write BNF grammars exactly the way we'd like, but we can come reasonably close. Moreover, I think it's fun to use  $TLA^+$  to specify its own syntax. So, module *BNFGrammars* is used in Chapter 15 to specify part of the syntax of  $TLA^+$ , as well as in Chapter 14 to specify the syntax of the TLC model checker's configuration file.

Let's start by reviewing BNF grammars. Consider the little language *SE* of simple expressions described by the BNF grammar

```
expr ::= ident | expr op expr | (expr) | LET def IN expr
def ::= ident == expr
```

where **op** is some class of infix operators like  $+$ , and **ident** is some class of identifiers such as *abc* and *x*. The language *SE* contains expressions like

*abc*  $+$  (LET *x* == *y* + *abc* IN *x* \* *x*)

Let's represent this expression as the sequence

```
("abc", "+", "(", "LET", "x", "==",
 "y", "+", "abc", "IN", "x", "*", "x", ")")
```

of strings. The strings such as "abc" and "+" appearing in this sequence are usually called *lexemes*. In general, a sequence of lexemes is called a *sentence*; and a set of sentences is called a *language*. So, we want to define the language *SE* to consist of the set of all such sentences described by the BNF grammar.<sup>3</sup>

To represent a BNF grammar in  $TLA^+$ , we must assign a mathematical meaning to nonterminal symbols like *def*, to terminal symbols like **op**, and to the grammar's two productions. The method that I find simplest is to let the meaning of a nonterminal symbol be the language that it generates. Thus, the meaning of *expr* is the language *SE* itself. I define a *grammar* to be a function *G* such that, for any string "str", the value of *G*["str"] is the language generated by the nonterminal *str*. Thus, if *G* is the BNF grammar above, then *G*["expr"] is the complete language *SE*, and *G*["def"] is the language defined by the production for *def*, which contains sentences like

```
("y", "==" , "qq", "*", "wxyz")
```

---

<sup>3</sup>BNF grammars are also used to specify how an expression is parsed—for example, that  $a + b * c$  is parsed as  $a + (b * c)$  rather than  $(a + b) * c$ . By considering the grammar to specify only a set of sentences, we are deliberately not capturing that use in our  $TLA^+$  representation of BNF grammars.

Instead of letting the domain of  $G$  consist of just the two strings “expr” and “def”, it turns out to be more convenient to let its domain be the entire set STRING of strings, and to let  $G[s]$  be the empty language (the empty set) for all strings  $s$  other than “expr” and “def”. So, a grammar is a function from the set of all strings to the set of sequences of strings. We can therefore define the set  $\text{Grammar}$  of all grammars by

$$\text{Grammar} \triangleq [\text{STRING} \rightarrow \text{SUBSET } \text{Seq}(\text{STRING})]$$

In describing the mathematical meaning of records, Section 5.2 explained that  $r.ack$  is an abbreviation for  $r[“ack”]$ . This is the case even if  $r$  isn’t a record. So, we can write  $G.op$  instead of  $G[“op”]$ . (A grammar isn’t a record because its domain is the set of all strings rather than a finite set of strings.)

A terminal like **ident** can appear anywhere to the right of a “ $::=$ ” that a nonterminal like *expr* can, so a terminal should also be a set of sentences. Let’s represent a terminal as a set of sentences, each of which is a sequence consisting of a single lexeme. Let a *token* be a sentence consisting of a single lexeme, so a terminal is a set of tokens. For example, the terminal **ident** is a set containing tokens such as  $\langle “abc” \rangle$ ,  $\langle “x” \rangle$ , and  $\langle “qq” \rangle$ . Any terminal appearing in the BNF grammar must be represented by a set of tokens, so the  $==$  in the grammar for SE is the set  $\{ \langle “==” \rangle \}$ . Let’s define the operator *tok* by

$$\text{tok}(s) \triangleq \{ \langle s \rangle \}$$

so we can write this set of tokens as  $\text{tok}(“==”)$ .

A production expresses a relation between the values of  $G.str$  for some grammar  $G$  and some strings “str”. For example, the production

$$\text{def} ::= \text{ident} == \text{expr}$$

asserts that a sentence  $s$  is in  $G.\text{def}$  iff it has the form  $i \circ \langle “==” \rangle \circ e$  for some token  $i$  in **ident** and some sentence  $e$  in  $G.\text{expr}$ . In mathematics, a formula about  $G$  must mention  $G$  (perhaps indirectly by using a symbol defined in terms of  $G$ ). So, we can try writing this production in TLA<sup>+</sup> as

$$G.\text{def} ::= \text{ident} \text{ tok}(“==”) \text{ } G.\text{expr}$$

In the expression to the right of the  $::=$ , adjacency is expressing some operation. Just as we have to make multiplication explicit by writing  $2 * x$  instead of  $2x$ , we must express this operation by an explicit operator. Let’s use  $\&$ , so we can write the production as

$$(11.1) \quad G.\text{def} ::= \text{ident} \& \text{ tok}(“==”) \& G.\text{expr}$$

This expresses the desired relation between the sets  $G.\text{def}$  and  $G.\text{expr}$  of sentences if  $::=$  is defined to be equality and  $\&$  is defined so that  $L \& M$  is the

*tok* is short for  
*token*.

set of all sentences obtained by concatenating a sentence in  $L$  with a sentence in  $M$ :

$$L \& M \triangleq \{s \circ t : s \in L, t \in M\}$$

The production

$$\text{expr} ::= \text{ident} \mid \text{expr op expr} \mid (\text{expr}) \mid \text{LET def IN expr}$$

can similarly be expressed as

$$(11.2) \quad G.\text{expr} ::= \begin{array}{l} \text{ident} \\ | \quad G.\text{expr} \& \text{op} \& G.\text{expr} \\ | \quad \text{tok}(“(“) \& G.\text{expr} \& \text{tok}(“)”“) \\ | \quad \text{tok}(“LET”) \& G.\text{def} \& \text{tok}(“IN”) \& G.\text{expr} \end{array}$$

The precedence rules of  $\text{TLA}^+$  imply that  $a \mid b \& c$  is interpreted as  $a \mid (b \& c)$ .

This expresses the desired relation if  $\mid$  (which means *or* in the BNF grammar) is defined to be set union ( $\cup$ ).

We can also define the following operators that are sometimes used in BNF grammars:

- $\text{Nil}$  is defined so that  $\text{Nil} \& S$  equals  $S$  for any set  $S$  of sentences:

$$\text{Nil} \triangleq \{\langle \rangle\}$$

- $L^+$  equals  $L \mid L \& L \mid L \& L \& L \mid \dots$ :

$$L^+ \triangleq \text{LET } LL[n \in \text{Nat}] \triangleq \underbrace{LL[n] = L \mid \dots \mid \overbrace{L \& \dots L}^{n+1 \text{ copies}}}_{\begin{array}{l} \text{IF } n = 0 \text{ THEN } L \\ \text{ELSE } LL[n-1] \mid LL[n-1] \& L \\ \text{IN UNION } \{LL[n] : n \in \text{Nat}\} \end{array}}$$

$L^+$  is typed  $\text{L}^+$  and  $L^*$  is typed  $\text{L}^*$ .

- $L^*$  equals  $\text{Nil} \mid L \mid L \& L \mid L \& L \& L \mid \dots$ :

$$L^* \triangleq \text{Nil} \mid L^+$$

The BNF grammar for SE consists of two productions, expressed by the  $\text{TLA}^+$  formulas (11.1) and (11.2). The entire grammar is the single formula that is the conjunction of these two formulas. We must turn this formula into a mathematical definition of a grammar  $GSE$ , which is a function from strings to languages. The formula is an assertion about a grammar  $G$ . We define  $GSE$  to be the smallest grammar  $G$  satisfying the conjunction of (11.1) and (11.2), where grammar  $G_1$  smaller than  $G_2$  means that  $G_1[s] \subseteq G_2[s]$  for every string  $s$ . To express this in  $\text{TLA}^+$ , we define an operator  $\text{LeastGrammar}$  so that  $\text{LeastGrammar}(P)$  is the smallest grammar  $G$  satisfying  $P(G)$ :

$$\text{LeastGrammar}(P(\_)) \triangleq$$

CHOOSE  $G \in \text{Grammar} :$

$$\wedge P(G)$$

$$\wedge \forall H \in \text{Grammar} : P(H) \Rightarrow (\forall s \in \text{STRING} : G[s] \subseteq H[s])$$

Letting  $P(G)$  be the conjunction of (11.1) and (11.2), we can define the grammar  $GSE$  to be  $\text{LeastGrammar}(P)$ . We can then define the language  $\text{SE}$  to equal  $GSE.\text{expr}$ . The smallest grammar  $G$  satisfying a formula  $P$  must have  $G[s]$  equal to the empty language for any string  $s$  that doesn't appear in  $P$ . Thus,  $GSE[s]$  equals the empty language  $\{\}$  for any string  $s$  other than “expr” and “def”.

To complete our specification of  $GSE$ , we must define the sets  $\text{ident}$  and  $\text{op}$  of tokens. We can define the set  $\text{op}$  of operators by enumerating them—for example:

$$\text{op} \triangleq \text{tok}(“+”) \mid \text{tok}(“-”) \mid \text{tok}(“*”) \mid \text{tok}(“/”)$$

To express this a little more compactly, let's define  $\text{Tok}(S)$  to be the set of all tokens formed from elements in the set  $S$  of lexemes:

$$\text{Tok}(S) \triangleq \{\langle s \rangle : s \in S\}$$

We can then write

$$\text{op} \triangleq \text{Tok}(\{“+”, “-”, “*”, “/”\})$$

Let's define  $\text{ident}$  to be the set of tokens whose lexemes are words made entirely of lower-case letters, such as “abc”, “qq”, and “x”. To learn how to do that, we must first understand what strings in  $\text{TLA}^+$  really are. In  $\text{TLA}^+$ , a string is a sequence of characters. (We don't care, and the semantics of  $\text{TLA}^+$  doesn't specify, what a character is.) We can therefore apply the usual sequence operators on them. For example,  $\text{Tail}(“abc”)$  equals “bc”, and “abc”  $\circ$  “de” equals “abcde”.

The operators like  $\&$  that we just defined for expressing BNF were applied to sets of sentences, where a sentence is a sequence of lexemes. These operators can be applied just as well to sets of sequences of any kind—including sets of strings. For example,  $\{\text{“one”}, \text{“two”}\} \& \{\text{“s”}\}$  equals  $\{\text{“ones”}, \text{“twos”}\}$ , and  $\{\text{“ab”}\}^+$  is the set consisting of all the strings “ab”, “abab”, “ababab”, etc. So, we can define  $\text{ident}$  to equal  $\text{Tok}(\text{Letter}^+)$ , where  $\text{Letter}$  is the set of all lexemes consisting of a single lower-case letter:

$$\text{Letter} \triangleq \{\text{“a”}, \text{“b”}, \dots, \text{“z”}\}$$

Writing this definition out in full (without the “...”) is tedious. We can make this a little easier as follows. We first define  $\text{OneOf}(s)$  to be the set of all one-character strings made from the characters of the string  $s$ :

$$\text{OneOf}(s) \triangleq \{\langle s[i] \rangle : i \in \text{DOMAIN } s\}$$

We can then define

$$\text{Letter} \triangleq \text{OneOf}(“abcdefghijklmnopqrstuvwxyz”)$$

See Section 16.1.10 on page 307 for more about strings. Remember that we take *sequence* and *tuple* to be synonymous.

$$\begin{aligned}
 GSE &\triangleq \text{LET } op \triangleq \text{Tok}(\{ "+", "-", "*", "/" \}) \\
 &\quad ident \triangleq \text{Tok}(\text{OneOf}("abcdefghijklmnopqrstuvwxyz")^+) \\
 P(G) &\triangleq \wedge G.\text{expr} ::= \begin{array}{l} ident \\ | G.\text{expr} \& op \& G.\text{expr} \\ | tok("(") \& G.\text{expr} \& tok(")") \\ | tok("LET") \& G.\text{def} \& tok("IN") \& G.\text{expr} \end{array} \\
 &\quad \wedge G.\text{def} ::= ident \& tok("==") \& G.\text{expr} \\
 \text{IN } &LeastGrammar(P)
 \end{aligned}$$

**Figure 11.3:** The definition of the grammar  $GSE$  for the language SE.

The complete definition of the grammar  $GSE$  appears in Figure 11.3 on this page.

All the operators we've defined here for specifying grammars are grouped into module *BNFGrammars*, which appears in Figure 11.4 on the next page.

Using  $TLA^+$  to write ordinary BNF grammars is a bit silly. However, ordinary BNF grammars are not very convenient for describing the syntax of a complicated language like  $TLA^+$ . In fact, they can't describe the alignment rules for its bulleted lists of conjuncts and disjuncts. Using  $TLA^+$  to specify such a language is not so silly. In fact, a  $TLA^+$  specification of the complete syntax of  $TLA^+$  was written as part of the development of the Syntactic Analyzer, described in Chapter 12. Although valuable when writing a  $TLA^+$  parser, this specification isn't very helpful to an ordinary user of  $TLA^+$ , so it does not appear in this book.

## 11.2 Other Memory Specifications

Section 5.3 specifies a multiprocessor memory. The specification is unrealistically simple for three reasons: a processor can have only one outstanding request at a time, the basic correctness condition is too restrictive, and only simple read and write operations are provided. (Real memories provide many other operations, such as partial-word writes and cache prefetches.) We now specify a memory that allows multiple outstanding requests and has a realistic, weaker correctness condition. To keep the specification short, we still consider only the simple operations of reading and writing one word of memory.

### 11.2.1 The Interface

The first thing we must do to specify a memory is determine the interface. The interface we choose depends on the purpose of the specification. There are many different reasons why we might be specifying a multiprocessor memory. We could

MODULE *BNFGrammars*

A sentence is a sequence of strings. (In standard terminology, the term “lexeme” is used instead of “string”.) A token is a sentence of length one—that is, a one-element sequence whose single element is a string. A language is a set of sentences.

LOCAL INSTANCE *Naturals*

LOCAL INSTANCE *Sequences*

## OPERATORS FOR DEFINING SETS OF TOKENS

$$OneOf(s) \triangleq \{\langle s[i] \rangle : i \in \text{DOMAIN } s\}$$

If  $s$  is a string, then  $OneOf(s)$  is the set of strings formed from the individual characters of  $s$ . For example,  $OneOf("abc") = \{"a", "b", "c"\}$ .

$$tok(s) \triangleq \{\langle s \rangle\}$$

If  $s$  is a string, then  $tok(s)$  is the set containing only the

$$Tok(S) \triangleq \{\langle s \rangle : s \in S\}$$

token made from  $s$ . If  $S$  is a set of strings, then  $Tok(S)$  is

the set of tokens made from elements of  $S$ .

## OPERATORS FOR DEFINING LANGUAGES

$$Nil \triangleq \{\langle \rangle\} \quad \text{The language containing only the “empty” sentence.}$$

$$L \& M \triangleq \{s \circ t : s \in L, t \in M\} \quad \text{All concatenations of sentences in } L \text{ and } M.$$

$$L \mid M \triangleq L \cup M$$

$$L^+ \triangleq L \mid L \& L \mid L \& L \& L \mid \dots$$

$$\begin{aligned} \text{LET } LL[n \in \text{Nat}] \triangleq & \text{ IF } n = 0 \text{ THEN } L \\ & \text{ELSE } LL[n - 1] \mid LL[n - 1] \& L \end{aligned}$$

$$\text{IN } \text{UNION } \{LL[n] : n \in \text{Nat}\}$$

$$L^* \triangleq Nil \mid L^+$$

$$L ::= M \triangleq L = M$$

$$Grammar \triangleq [\text{STRING} \rightarrow \text{SUBSET } Seq(\text{STRING})]$$

$$LeastGrammar(P(\_)) \triangleq \text{ The smallest grammar } G \text{ such that } P(G) \text{ is true.}$$

CHOOSE  $G \in Grammar$  :

$$\wedge P(G)$$

$$\wedge \forall H \in Grammar : P(H) \Rightarrow \forall s \in \text{STRING} : G[s] \subseteq H[s]$$

Figure 11.4: The module *BNFGrammars*.

be specifying a computer architecture, or the semantics of a programming language. Let's suppose we are specifying the memory of an actual multiprocessor computer.

A modern processor performs multiple instructions concurrently. It can begin new memory operations before previous ones have been completed. The memory responds to a request as soon as it can; it need not respond to different requests in the order that they were issued.

A processor issues a request to a memory system by setting some register. We assume that each processor has a set of registers through which it communicates with the memory. Each register has three fields: an *adr* field that holds an address, a *val* field that holds a word of memory, and an *op* field that indicates what kind of operation, if any, is in progress. The processor can issue a command using a register whose *op* field equals “Free”. It sets the *op* field to “Rd” or “Wr” to indicate the operation; it sets the *adr* field to the address of the memory word; and, for a write, it sets the *val* field to the value being written. (On a read, the processor can set the *val* field to any value.) The memory responds by setting the *op* field back to “Free” and, for a read, setting the *val* field to the value read. (The memory does not change the *val* field when responding to a write.)

Module *RegisterInterface* in Figure 11.5 on the next page contains some declarations and definitions for specifying the interface. It declares the constants *Adr*, *Val*, and *Proc*, which are the same as in the memory interface of Section 5.1, and the constant *Reg*, which is the set of registers. (More precisely, *Reg* is a set of register identifiers.) A processor has a separate register corresponding to each element of *Reg*. The variable *regFile* represents the processors' registers, *regFile*[*p*][*r*] being register *r* of processor *p*. The module also defines the sets of requests and register values, as well as a type invariant for *regFile*.

### 11.2.2 The Correctness Condition

Section 5.3 specifies what is called a linearizable memory. In a linearizable memory, a processor never has more than one outstanding request. The correctness condition for the memory can be stated as

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and each operation is executed between the request and the response.

The second clause, which requires the system to act as if each operation were executed between its request and its response, is both too weak and too strong for our specification. It's too weak because it says nothing about the execution order of two operations from the same processor unless one is issued after the other's response. For example, suppose a processor *p* issues a write and then a read to the same address. We want the read to obtain either the value *p* just wrote, or a value written by another processor—even if *p* issues the read before

| MODULE <i>RegisterInterface</i>                                                         |                                                                                               |
|-----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| CONSTANT <i>Adr</i> ,                                                                   | The set of memory addresses.                                                                  |
| <i>Val</i> ,                                                                            | The set of memory-word values.                                                                |
| <i>Proc</i> ,                                                                           | The set of processors.                                                                        |
| <i>Reg</i>                                                                              | The set of registers used by a processor.                                                     |
| VARIABLE <i>regFile</i>                                                                 | $\text{regFile}[p][r]$ represents the contents of register <i>r</i> of processor <i>p</i> .   |
| <i>RdRequest</i>                                                                        | $\triangleq [adr : \text{Adr}, val : \text{Val}, op : \{\text{"Rd"}\}]$                       |
| <i>WrRequest</i>                                                                        | $\triangleq [adr : \text{Adr}, val : \text{Val}, op : \{\text{"Wr"}\}]$                       |
| <i>FreeRegValue</i>                                                                     | $\triangleq [adr : \text{Adr}, val : \text{Val}, op : \{\text{"Free"}\}]$                     |
| <i>Request</i>                                                                          | $\triangleq \text{RdRequest} \cup \text{WrRequest}$ The set of all possible requests.         |
| <i>RegValue</i>                                                                         | $\triangleq \text{Request} \cup \text{FreeRegValue}$ The set of all possible register values. |
| <i>RegFileTypeInvariant</i>                                                             | $\triangleq$ The type correctness invariant for <i>regFile</i> .                              |
| $\text{regFile} \in [\text{Proc} \rightarrow [\text{Reg} \rightarrow \text{RegValue}]]$ |                                                                                               |

**Figure 11.5:** A module for specifying a register interface to a memory.

receiving the response for the write. This isn't guaranteed by the condition. The second clause is too strong because it places unnecessary ordering constraints on operations issued by different processors. If operations *A* and *B* are issued by two different processors, then we don't need to require that *A* precedes *B* in the execution order just because *B* was requested after *A*'s response.

We modify the second clause to require that the system act as if operations of each individual processor were executed in the order that they were issued, obtaining the condition

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order in which the requests were issued.

In other words, we require that the values returned by the reads can be explained by some total ordering of the operation executions that is consistent with the order in which each processor issued its requests. There are a number of different ways of formalizing this condition; they differ in how bizarre the explanation may be. The differences can be described in terms of whether or not certain scenarios are permitted. In the scenario descriptions,  $Wr_p(a, v)$  represents a write operation of value *v* to address *a* by processor *p*, and  $Rd_p(a, v)$  represents a read of *a* by *p* that returns the value *v*.

The first decision we must make is whether all operations in an infinite behavior must be ordered, or if the ordering must exist only at each finite point

during the behavior. Consider a scenario in which each of two processors writes its own value to the same address and then keeps reading that value forever:

Processor  $p$ :  $Wr_p(a, v1)$ ,  $Rd_p(a, v1)$ ,  $Rd_p(a, v1)$ ,  $Rd_p(a, v1)$ ,  $\dots$   
 Processor  $q$ :  $Wr_q(a, v2)$ ,  $Rd_q(a, v2)$ ,  $Rd_q(a, v2)$ ,  $Rd_q(a, v2)$ ,  $\dots$

In these scenarios, values and addresses with different names are assumed to be different.

At each point in the execution, we can explain the values returned by the reads with a total order in which all the operations of either processor precede all the operations of the other. However, there is no way to explain the entire infinite scenario with a single total order. In this scenario, neither processor ever sees the value written by the other. Since a multiprocessor memory is supposed to allow processors to communicate, we disallow this scenario.

The second decision we must make is whether the memory is allowed to predict the future. Consider this scenario:

Processor  $p$ :  $Wr_p(a, v1)$ ,  $Rd_p(a, v2)$   
 Processor  $q$ :  $Wr_q(a, v2)$

Here,  $q$  issues its write of  $v2$  after  $p$  has obtained the result of its read. The scenario is explained by the ordering  $Wr_p(a, v1)$ ,  $Wr_q(a, v2)$ ,  $Rd_p(a, v2)$ . However, this is a bizarre explanation because, to return the value  $v2$  for  $p$ 's read, the memory had to predict that another processor would write  $v2$  some time in the future. Since a real memory can't predict what requests will be issued in the future, such a behavior cannot be produced by a correct implementation. We can therefore rule out the scenario as unreasonable. Alternatively, since no correct implementation can produce it, there's no need to outlaw the scenario.

If we don't allow the memory to predict the future, then it must always be able to explain the values read in terms of the writes that have been issued so far. In this case, we have to decide whether the explanations must be stable. For example, suppose a scenario begins as follows:

Processor  $p$ :  $Wr_p(a1, v1)$ ,  $Rd_p(a1, v3)$   
 Processor  $q$ :  $Wr_q(a2, v2)$ ,  $Wr_q(a1, v3)$

At this point, the only explanation for  $p$ 's read  $Rd_p(a1, v3)$  is that  $q$ 's write  $Wr_q(a1, v3)$  preceded it, which implies that  $q$ 's other write  $Wr_q(a2, v2)$  also preceded the read. Hence, if  $p$  now reads  $a2$ , it must obtain the value  $v2$ . But suppose the scenario continues as follows, with another processor  $r$  joining in:

Processor  $p$ :  $Wr_p(a1, v1)$ ,  $Rd_p(a1, v3)$ ,  $Rd_p(a2, v0)$   
 Processor  $q$ :  $Wr_q(a2, v2)$ ,  $Wr_q(a1, v3)$   
 Processor  $r$ :  $Wr_r(a1, v3)$

We can explain this scenario with the following ordering of the operations:

$Wr_p(a1, v1)$ ,  $Wr_r(a1, v3)$ ,  $Rd_p(a1, v3)$ ,  
 $Rd_p(a2, v0)$ ,  $Wr_q(a2, v2)$ ,  $Wr_q(a1, v3)$

In this explanation, processor  $r$  provided the value of  $a1$  read by  $p$ , and  $p$  read the initial value  $v0$  of memory address  $a2$ . The explanation is bizarre because the write that provided the value of  $a1$  to  $p$  was actually issued after the completion of  $p$ 's read operation. But, because the explanation of that value changed in mid-execution, the system never predicted the existence of a write that had not yet occurred. When writing a specification, we must decide whether or not to allow such changes of the explanation.

### 11.2.3 A Serial Memory

We first specify a memory that cannot predict the future and cannot change its explanations. There seems to be no standard name for such a memory; I'll call it a *serial* memory.

Our informal correctness condition is in terms of the sequence of all operations that have ever been issued. There is a general method of formalizing such a condition that works for specifying many different kinds of systems. We add an internal variable  $opQ$  that records the history of the execution. For each processor  $p$ , the value of  $opQ[p]$  is a sequence whose  $i^{\text{th}}$  element,  $opQ[p][i]$ , describes the  $i^{\text{th}}$  request issued by  $p$ , the response to that request (if it has been issued), and any other information about the operation needed to express the correctness condition. If necessary, we can also add other internal variables to record information not readily associated with individual requests.

For a system with the kind of register interface we are using, the next-state action has the form

$$\begin{aligned}
 (11.3) \vee \exists proc \in Proc, reg \in Reg : \\
 & \vee \exists req \in Request : IssueRequest(proc, req, reg) \\
 & \vee RespondToRequest(proc, reg) \\
 & \vee Internal
 \end{aligned}$$

where the component actions are

*IssueRequest*( $proc, req, reg$ )

The action with which processor  $proc$  issues a request  $req$  in register  $reg$ .

*RespondToRequest*( $proc, reg$ )

The action with which the system responds to a request in processor  $proc$ 's register  $reg$ .

*Internal*

An action that changes only the internal state.

Liveness properties are asserted by fairness conditions on the *RespondToRequest* and *Internal* actions.

A general trick for writing the specification is to choose the internal state so the safety part of the correctness condition can be expressed by the formula  $\Box P$  for some state predicate  $P$ . We guarantee that  $P$  is always true by letting  $P'$  be a conjunct of each action. I'll use this approach to specify the serial memory, taking for  $P$  a state predicate *Serializable*.

We want to require that the value returned by each read is explainable as the value written by some operation already issued, or as the initial value of the memory. Moreover, we don't want this explanation to change. We therefore add to the  $opQ$  entry for each completed read a *source* field that indicates where the value came from. This field is set by the *RespondToRequest* action.

We want all operations in an infinite behavior eventually to be ordered. This means that, for any two operations, the memory must eventually decide which one precedes the other—and it must stick to that decision. We introduce an internal variable  $opOrder$  that describes the ordering of operations to which the memory has already committed itself. An *Internal* step changes only  $opOrder$ , and it can only enlarge the ordering.

The predicate *Serializable* used to specify the safety part of the correctness condition describes what it means for  $opOrder$  to be a correct explanation. It asserts that there is some consistent total ordering of the operations that satisfies the following conditions:

- It extends  $opOrder$ .
- It orders all operations from the same processor in the order that they were issued.
- It orders operations so that the source of any read is the latest write to the same address that precedes the read, and is the initial value iff there is no such write.

We now translate this informal sketch of the specification into TLA<sup>+</sup>. We first choose the types of the variables  $opQ$  and  $opOrder$ . To do this, we define a set  $opId$  of values that identify the operations that have been issued. An operation is identified by a pair  $\langle p, i \rangle$  where  $p$  is a processor and  $i$  is a position in the sequence  $opQ[p]$ . (The set of all such positions  $i$  is  $\text{DOMAIN } opQ[p]$ .) We let the corresponding element of  $opId$  be the record with *proc* field  $p$  and *idx* field  $i$ . Writing the set of all such records is a bit tricky because the possible values of the *idx* field depend on the *proc* field. We define  $opId$  to be a subset of the set of records whose *idx* field can be any value:

$$opId \triangleq \{ oiv \in [proc : Proc, idx : Nat] : \\ oiv.idx \in \text{DOMAIN } opQ[oiv.proc] \}$$

For convenience, we define  $opIdQ(oi)$  to be the value of the  $opQ$  entry identified by an element  $oi$  of  $opId$ :

$$opIdQ(oi) \triangleq opQ[oi.proc][oi.idx]$$

The source of a value need not be an operation; it can also be the initial contents of the memory. The latter possibility is represented by letting the *source* field of the  $opQ$  entry have the special value  $InitWr$ . We then let  $opQ$  be an element of  $[Proc \rightarrow Seq(opVal)]$ , where  $opVal$  is the union of three sets:

$[req : Request, reg : Reg]$

Represents an active request in the register of the requesting processor indicated by the  $reg$  field.

The sets  $Request$ ,  $WrRequest$ , and  $RdRequest$  are defined in module *RegisterInterface* on page 186.

$[req : WrRequest, reg : \{Done\}]$

Represents a completed write request, where  $Done$  is a special value that is not a register.

$[req : RdRequest, reg : \{Done\}, source : opId \cup \{InitWr\}]$

Represents a completed read request whose value came from the operation indicated by the *source* field, or from the initial value of the memory location if the *source* field equals  $InitWr$ .

Note that  $opId$  and  $opVal$  are state functions whose values depend upon the value of the variable  $opQ$ .

We need to specify the initial contents of memory. A program generally cannot assume anything about the memory’s initial contents, except that every address does contain a value in  $Val$ . So, the initial contents of memory can be any element of  $[Adr \rightarrow Val]$ . We declare an “internal” constant  $InitMem$ , whose value is the memory’s initial contents. In the final specification,  $InitMem$  will be hidden along with the internal variables  $opQ$  and  $opOrder$ . We hide a constant with ordinary existential quantification  $\exists$ . The requirement that  $InitMem$  is a function from addresses to values could be made part of the initial predicate, but it’s more natural to express it in the quantifier. The final specification will therefore have the form

$\exists InitMem \in [Adr \rightarrow Val] : \exists opQ, opOrder : \dots$

For later use, we define  $goodSource(oi)$  to be the set of plausible values for the source of a read operation  $oi$  in  $opId$ . A plausible value is either  $InitWr$  or a write to the same address that  $oi$  reads. It will be an invariant of the specification that the source of any completed read operation  $oi$  is an element of  $goodSource(oi)$ . Moreover, the value returned by a completed read operation must come from its source. If the source is  $InitWr$ , then the value must come from  $InitMem$ ; otherwise, it must come from the source request’s *val* field. To express this formally, observe that the  $opQ$  entries only of completed reads have a *source* field. Since a record has a *source* field iff the string “source” is in its domain, we can write this invariant as

Section 5.2 on page 48 explains that a record is a function whose domain is a set of strings.

(11.4)  $\forall oi \in opId :$

$$\begin{aligned}
 & (\text{"source"} \in \text{DOMAIN } opIdQ(oi)) \Rightarrow \\
 & \quad \wedge \ opIdQ(oi).\text{source} \in \text{goodSource}(oi) \\
 & \quad \wedge \ opIdQ(oi).\text{req.val} = \text{IF } opIdQ(oi).\text{source} = \text{InitWr} \\
 & \quad \quad \quad \text{THEN } InitMem[opIdQ(oi).\text{req.adr}] \\
 & \quad \quad \quad \text{ELSE } opIdQ(opIdQ(oi).\text{source}).\text{req.val}
 \end{aligned}$$

We now choose the type of  $opOrder$ . We usually denote an ordering relation by an operator such as  $\prec$ , writing  $A \prec B$  to mean that  $A$  precedes  $B$ . However, the value of a variable cannot be an operator. So, we must represent an ordering relation as a set or a function. Mathematicians usually describe a relation  $\prec$  on a set  $S$  as a set  $R$  of ordered pairs of elements in  $S$ , with  $\langle A, B \rangle$  in  $R$  iff  $A \prec B$ . So, we let  $opOrder$  be a subset of  $opId \times opId$ , where  $\langle oi, oj \rangle \in opOrder$  means that  $oi$  precedes  $oj$ .

The difference between operators and functions is discussed in Section 6.4 on page 69.

Our internal state is redundant because, if register  $r$  of processor  $p$  contains an uncompleted operation, then there is an  $opQ$  entry that points to the register and contains the same request. This redundancy means that the following relations among the variables are invariants of the specification:

- If an  $opQ$  entry's  $reg$  field is not equal to  $Done$ , then it denotes a register whose contents is the entry's  $req$  field.
- The number of  $opQ$  entries pointing to a register equals 1 if the register contains an active operation, otherwise it equals 0.

In the specification, we combine this condition, formula (11.4), and the type invariant into a single state predicate *DataInvariant*.

Having chosen the types of the variables, we can now define the initial predicate *Init* and the predicate *Serializable*. The definition of *Init* is easy. We define *Serializable* in terms of *totalOpOrder*, the set of all total orders of  $opId$ . A relation  $\prec$  is a total order of  $opId$  iff it satisfies the following three conditions, for any  $oi, oj$ , and  $ok$  in  $opId$ :

Totality: Either  $oi = oj$ ,  $oi \prec oj$ , or  $oj \prec oi$ .

Transitivity:  $oi \prec oj$  and  $oj \prec ok$  imply  $oi \prec ok$ .

Irreflexivity:  $oi \not\prec oi$ .

The predicate *Serializable* asserts that there is a total order of  $opId$  satisfying the three conditions on page 189. We can express this formally as the assertion that there exists an  $R$  in *totalOpOrder* satisfying

$$\wedge \ opOrder \subseteq R$$

$R$  extends *opOrder*

$$\wedge \ \forall oi, oj \in opId :$$

$R$  correctly orders operations from the same processor.

$$(oi.\text{proc} = oj.\text{proc}) \wedge (oi.\text{idx} < oj.\text{idx}) \Rightarrow (\langle oi, oj \rangle \in R)$$

$$\begin{aligned}
 & \wedge \forall oi \in opId : \\
 & \quad (\text{"source"} \in \text{DOMAIN } opIdQ(oi)) \quad \text{For every completed read } oi \text{ in } opId, \text{ there is no write} \\
 & \quad \Rightarrow \neg (\exists oj \in \text{goodSource}(oi) : \quad \text{ } \\
 & \quad \quad \wedge \langle oj, oi \rangle \in R \quad \text{ } \\
 & \quad \quad \wedge (opIdQ(oi).\text{source} \neq \text{InitWr}) \Rightarrow \\
 & \quad \quad \quad (\langle opIdQ(oi).\text{source}, oj \rangle \in R) )
 \end{aligned}$$

We allow each step to extend  $opOrder$  to any relation on  $opId$  that satisfies *Serializable*. We do this by letting every subaction of the next-state action specify  $opOrder'$  with the conjunct  $UpdateOpOrder$ , defined by

$$\begin{aligned}
 UpdateOpOrder \triangleq & \wedge opOrder' \subseteq (opId' \times opId') \\
 & \wedge opOrder \subseteq opOrder' \\
 & \wedge Serializable'
 \end{aligned}$$

The next-state action has the generic form of formula (11.3) on page 188. We split the *RespondToRequest* action into the disjunction of separate *RespondToWr* and *RespondToRd* actions that represent responding to writes and reads, respectively. *RespondToRd* is the most complicated of the next-state action's subactions, so let's examine its definition. The definition has the form

$$\begin{aligned}
 RespondToRd(proc, reg) \triangleq & \\
 \text{LET } req \triangleq & \text{regFile}[proc][reg] \\
 \text{idx} \triangleq & \text{CHOOSE } i \in \text{DOMAIN } opQ[proc] : opQ[proc][i].reg = reg \\
 \text{IN } \dots
 \end{aligned}$$

This defines  $req$  to be the request in the register and  $idx$  to be an element in the domain of  $opQ[proc]$  such that  $opQ[proc][idx].reg$  equals  $reg$ . If the register is not free, then there is exactly one such value  $idx$ ; and  $opQ[proc][idx].req$ , the  $idx^{\text{th}}$  request issued by  $proc$ , equals  $req$ . (We don't care what  $idx$  equals if the register is free.) The IN expression begins with the enabling condition

$$\wedge req.op = \text{"Rd"}$$

which asserts that the register is not free and it contains a read request. The next conjunct of the IN expression is

$$\begin{aligned}
 & \wedge \exists src \in \text{goodSource}([proc \mapsto proc, idx \mapsto idx]) : \\
 & \quad \text{LET } val \triangleq \text{IF } src = \text{InitWr} \text{ THEN } \text{InitMem}[req.adr] \\
 & \quad \quad \quad \text{ELSE } opIdQ(src).req.val \\
 & \quad \text{IN } \dots
 \end{aligned}$$

It asserts the existence of a value  $src$ , which will be the source of the value returned by the read; and it defines  $val$  to be that value. If the source is the initial contents of memory, then the value is obtained from *InitMem*; otherwise,

it is obtained from the source request's *val* field. The inner IN expression has two conjuncts that specify the values of *regFile'* and *opQ'*. The first conjunct asserts that the register's *val* field is set to *val* and its *op* field is set to "Free", indicating that the register is made free.

$$\wedge \text{regFile}' = [\text{regFile EXCEPT } ![\text{proc}][\text{reg}].\text{val} = \text{val}, \\ ![\text{proc}][\text{reg}].\text{op} = \text{"Free"}]$$

The second conjunct of the inner IN expression describes the new value of *opQ*. Only the *idx*<sup>th</sup> element of *opQ[proc]* is changed. It is set to a record whose *req* field is the same as the original request *req*, except that its *val* field is equal to *val*; whose *reg* field equals *Done*; and whose *source* field equals *src*.

$$\wedge \text{opQ}' = [\text{opQ EXCEPT } \\ ![\text{proc}][\text{idx}] = [\text{req} \mapsto [\text{req EXCEPT } !.\text{val} = \text{val}], \\ \text{reg} \mapsto \text{Done}, \\ \text{source} \mapsto \text{src}]]$$

Finally, the outer IN clause ends with the conjunct

$$\wedge \text{UpdateOpOrder}$$

that determines the value of *opOrder'*. It also implicitly determines the possible choices of the source of the read—that is, the value of *opQ'[proc][idx].source*. For some choices of this value allowed by the second outer conjunct, there will be no value of *opOrder'* satisfying *UpdateOpOrder*. The conjunct *UpdateOpOrder* rules out those choices for the source.

The definitions of the other subactions *IssueRequest*, *RespondToWr*, and *Internal* of the next-state action are simpler, and I won't explain them.

Having finished the initial predicate and the next-state action, we must determine the liveness conditions. The first condition is that the memory must eventually respond to every operation. The response to a request in register *reg* of processor *proc* is produced by a *RespondToWr(proc, reg)* or *RespondToRd(proc, reg)* action. So, the obvious way to express this condition is

$$\forall \text{proc} \in \text{Proc}, \text{reg} \in \text{Reg} : \\ \text{WF}_{\langle \dots \rangle}(\text{RespondToWr}(\text{proc}, \text{reg}) \vee \text{RespondToRd}(\text{proc}, \text{reg}))$$

For this fairness condition to imply that the response is eventually issued, a *RespondToWr(proc, reg)* or *RespondToRd(proc, reg)* step must be enabled whenever there is an uncompleted operation in *proc*'s register *reg*. It isn't completely obvious that a *RespondToRd(proc, reg)* step is enabled when there is a read operation in the register, since the step is enabled only if there exist a source for the read and a value of *opOrder'* that satisfy *Serializable'*. The required source and value do exist because *Serializable*, which holds in the first

state of the step, implies the existence of a correct total order of all the operations; this order can be used to choose a source and a relation  $opOrder'$  that satisfy *Serializability'*.

The second liveness condition asserts that the memory must eventually commit to an ordering for every pair of operations. It is expressed as a fairness condition, for every pair of distinct operations  $oi$  and  $oj$  in  $opId$ , on an *Internal* action that makes  $oi$  either precede or follow  $oj$  in the order  $opOrder'$ . A first attempt at this condition is

$$(11.5) \forall oi, oj \in opId : \\ (oi \neq oj) \Rightarrow WF_{\langle \dots \rangle}(\wedge Internal \\ \wedge (\langle oi, oj \rangle \in opOrder') \vee (\langle oj, oi \rangle \in opOrder'))$$

However, this isn't correct. In general, a formula  $\forall x \in S : F$  is equivalent to  $\forall x : (x \in S) \Rightarrow F$ . Hence, (11.5) is equivalent to the assertion that the following formula holds, for all constant values  $oi$  and  $oj$ :

$$(oi \in opId) \wedge (oj \in opId) \Rightarrow \\ \left( (oi \neq oj) \Rightarrow \\ \left( WF_{\langle \dots \rangle}(\wedge Internal \\ \wedge (\langle oi, oj \rangle \in opOrder') \vee (\langle oj, oi \rangle \in opOrder')) \right) \right)$$

In a temporal formula, a predicate with no temporal operators is an assertion about the initial state. Hence, (11.5) asserts that the fairness condition is true for all pairs of distinct values  $oi$  and  $oj$  in the initial value of  $opId$ . But  $opId$  is initially empty, so this condition is vacuously true. Hence, (11.5) is trivially implied by the initial predicate. We must instead assert fairness for the action

$$(11.6) \wedge (oi \in opId) \wedge (oj \in opId) \\ \wedge Internal \\ \wedge (\langle oi, oj \rangle \in opOrder') \vee (\langle oj, oi \rangle \in opOrder'))$$

for all distinct values  $oi$  and  $oj$ . It suffices to assert this only for  $oi$  and  $oj$  of the right type. Since it's best to use bounded quantifiers whenever possible, let's write this condition as

$$\forall oi, oj \in [proc : Proc, idx : Nat] : \quad \text{All operations are eventually ordered.} \\ (oi \neq oj) \Rightarrow WF_{\langle \dots \rangle}(\wedge (oi \in opId) \wedge (oj \in opId) \\ \wedge Internal \\ \wedge (\langle oi, oj \rangle \in opOrder') \vee (\langle oj, oi \rangle \in opOrder'))$$

For this formula to imply that any two operations are eventually ordered by  $opOrder$ , action (11.6) must be enabled if  $oi$  and  $oj$  are unordered operations in  $opId$ . It is, because *Serializable* is always enabled, so it is always possible to extend  $opOrder$  to a total order of all issued operations.

The complete inner specification, with *InitMem*, *opQ*, and *opOrder* visible, is in module *InnerSerial* on pages 196–198. I have made two minor modifications to allow the specification to be checked by the TLC model checker. (Chapter 14 describes TLC and explains why these changes are needed.) Instead of the definition of *opId* given on page 189, the specification uses the equivalent definition

$$opId \triangleq \text{UNION } \{ [proc : p, idx : \text{DOMAIN } opQ[p]] : p \in Proc \}$$

In the definition of *UpdateOpOrder*, the first conjunct is changed from

$$opOrder' \subseteq opId' \times opId'$$

to the equivalent

$$opOrder' \in \text{SUBSET } (opId' \times opId')$$

For TLC’s benefit, I also ordered the conjuncts of all actions so *UpdateOpOrder* follows the “assignment of a value to” *opQ'*. This resulted in the UNCHANGED conjunct not being the last one in action *Internal*.

The complete specification is written, as usual, with a parametrized instantiation of *InnerSerial* to hide the constant *InitMem* and the variables *opQ* and *opOrder*:

———— MODULE *SerialMemory* ————

EXTENDS *RegisterInterface*

*Inner(InitMem, opQ, opOrder)*  $\triangleq$  INSTANCE *InnerSerial*

*Spec*  $\triangleq$   $\exists \text{InitMem} \in [\text{Adr} \rightarrow \text{Val}] :$

$\exists \text{opQ, opOrder} : \text{Inner}(\text{InitMem}, \text{opQ}, \text{opOrder})! \text{Spec}$

### 11.2.4 A Sequentially Consistent Memory

The serial memory specification does not allow the memory to predict future requests. We now remove this restriction and specify what is called a *sequentially consistent* memory. The freedom to predict the future can’t be used by any real implementation,<sup>4</sup> so there’s little practical difference between a serial and a sequentially consistent memory. However, the sequentially consistent memory has a simpler specification. This specification is surprising and instructive.

The next-state action of the sequential memory specification has the same structure as that of the serial memory specification, with actions *IssueRequest*,

<sup>4</sup>The freedom to change explanations, which a sequentially consistent memory allows, could conceivably be used to permit a more efficient implementation, but it’s not easy to see how.

| MODULE <i>InnerSerial</i>                                                                                                                                                                                                                                                                                                                                                                                                                                     |  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| EXTENDS <i>RegisterInterface</i> , <i>Naturals</i> , <i>Sequences</i> , <i>FiniteSets</i>                                                                                                                                                                                                                                                                                                                                                                     |  |
| CONSTANT <i>InitMem</i> The initial contents of memory, which will be an element of $[Proc \rightarrow Adr]$ .                                                                                                                                                                                                                                                                                                                                                |  |
| VARIABLE $opQ$ , $opQ[p][i]$ is the $i^{\text{th}}$ operation issued by processor $p$ .                                                                                                                                                                                                                                                                                                                                                                       |  |
| $opOrder$ The order of operations, which is a subset of $opId \times opId$ . ( $opId$ is defined below).                                                                                                                                                                                                                                                                                                                                                      |  |
| $opId \triangleq \text{UNION } \{ [proc : \{p\}, idx : \text{DOMAIN } opQ[p] ] : p \in Proc \}$ $[proc \mapsto p, idx \mapsto i]$ identifies<br>$opIdQ(oi) \triangleq opQ[oi.proc][oi.idx]$ operation $i$ of processor $p$ .                                                                                                                                                                                                                                  |  |
| $InitWr \triangleq \text{CHOOSE } v : v \notin [proc : Proc, idx : Nat]$ The source for an initial memory value.                                                                                                                                                                                                                                                                                                                                              |  |
| $Done \triangleq \text{CHOOSE } v : v \notin Reg$ The $reg$ field value for a completed operation.                                                                                                                                                                                                                                                                                                                                                            |  |
| $opVal \triangleq$ Possible values of $opQ[p][i]$ .                                                                                                                                                                                                                                                                                                                                                                                                           |  |
| $\begin{aligned} & [req : Request, reg : Reg] && \text{An active request using register } reg \\ & \cup [req : WrRequest, reg : \{Done\}] && \text{A completed write.} \\ & \cup [req : RdRequest, reg : \{Done\}, source : opId \cup \{InitWr\}] && \text{A completed read of } source \text{ value.} \end{aligned}$                                                                                                                                         |  |
| $goodSource(oi) \triangleq$<br>$\{InitWr\} \cup \{o \in opId : \wedge opIdQ(o).req.op = \text{"Wr}$<br>$\wedge opIdQ(o).req.adr = opIdQ(oi).req.adr\}$                                                                                                                                                                                                                                                                                                        |  |
| $DataInvariant \triangleq$<br>$\wedge RegFileTypeInvariant$ Simple type invariants for $regFile$ ,<br>$\wedge opQ \in [Proc \rightarrow Seq(opVal)]$ $opQ$ , and<br>$\wedge opOrder \subseteq (opId \times opId)$ $opOrder$ .                                                                                                                                                                                                                                 |  |
| $\wedge \forall oi \in opId :$<br>$\wedge (\text{"source"} \in \text{DOMAIN } opIdQ(oi)) \Rightarrow$ The source of any completed read is either $InitWr$<br>$\wedge opIdQ(oi).source \in goodSource(oi)$ or a write operation to the same address.<br>$\wedge opIdQ(oi).req.val = \text{IF } opIdQ(oi).source = InitWr$ A read's value comes<br>$\text{THEN } InitMem[opIdQ(oi).req.adr]$ from its source.<br>$\text{ELSE } opIdQ(opIdQ(oi).source).req.val$ |  |
| $\wedge (opIdQ(oi).reg \neq Done) \Rightarrow$ $opQ$ correctly describes the register contents.<br>$(opIdQ(oi).req = regFile[oi.proc][opIdQ(oi).reg])$                                                                                                                                                                                                                                                                                                        |  |
| $\wedge \forall p \in Proc, r \in Reg :$ Only nonfree registers have corresponding $opQ$ entries.                                                                                                                                                                                                                                                                                                                                                             |  |
| $Cardinality(\{i \in \text{DOMAIN } opQ[p] : opQ[p][i].reg = r\}) =$<br>$\text{IF } regFile[p][r].op = \text{"Free"} \text{ THEN } 0 \text{ ELSE } 1$                                                                                                                                                                                                                                                                                                         |  |

Figure 11.6a: Module *InnerSerial* (beginning).

$Init \triangleq$  The initial predicate.

$\wedge regFile \in [Proc \rightarrow [Reg \rightarrow FreeRegValue]]$  Every register is free.

$\wedge opQ = [p \in Proc \mapsto \langle \rangle]$  There are no operations in  $opQ$ .

$\wedge opOrder = \{\}$  The order relation  $opOrder$  is empty.

$totalOpOrder \triangleq$  The set of all total orders on the set  $opId$ .

$\{R \in \text{SUBSET } (opId \times opId) :$

$\wedge \forall oi, oj \in opId : (oi = oj) \vee (\langle oi, oj \rangle \in R) \vee (\langle oj, oi \rangle \in R)$

$\wedge \forall oi, oj, ok \in opId : (\langle oi, oj \rangle \in R) \wedge (\langle oj, ok \rangle \in R) \Rightarrow (\langle oi, ok \rangle \in R)$

$\wedge \forall oi \in opId : \langle oi, oi \rangle \notin R\}$

$Serializable \triangleq$

$\exists R \in totalOpOrder : opOrder$ , asserts that there exists a total order  $R$  of all operations that extends  $opOrder$ , orders the operations of each processor correctly, and makes the source of each read the most recent write to the address.

$\wedge opOrder \subseteq R$

$\wedge \forall oi, oj \in opId : (oi.proc = oj.proc) \wedge (oi.idx < oj.idx) \Rightarrow (\langle oi, oj \rangle \in R)$

$\wedge \forall oi \in opId : (\text{"source"} \in \text{DOMAIN } opIdQ(oi)) \Rightarrow$

$\neg(\exists oj \in goodSource(oi) :$

$\wedge \langle oj, oi \rangle \in R$

$\wedge (opIdQ(oi).\text{source} \neq InitWr) \Rightarrow (\langle opIdQ(oi).\text{source}, oj \rangle \in R)$ )

↓

$UpdateOpOrder \triangleq$

$\wedge opOrder' \in \text{SUBSET } (opId' \times opId')$  An action that chooses the new value of  $opOrder$ , allowing it to be any relation that equals or extends the current value of  $opOrder$  and satisfies  $Serializable$ . This action is used in defining the subactions of the next-state action.

$\wedge opOrder \subseteq opOrder'$

$\wedge Serializable'$

$IssueRequest(proc, req, reg) \triangleq$

Processor  $proc$  issues request  $req$  in register  $reg$ .

$\wedge regFile[proc][reg].op = \text{"Free"}$  The register must be free.

$\wedge regFile' = [regFile \text{ EXCEPT } ![proc][reg] = req]$  Put the request in the register.

$\wedge opQ' = [opQ \text{ EXCEPT } ![proc] = Append(@, [req \mapsto req, reg \mapsto reg])]$  Add request to  $opQ[proc]$ .

$\wedge UpdateOpOrder$

$RespondToWr(proc, reg) \triangleq$  The memory responds to a write request in processor  $proc$ 's register  $reg$ .

$\wedge regFile[proc][reg].op = \text{"Wr"}$  The register must contain an active write request.

$\wedge regFile' = [regFile \text{ EXCEPT } ![proc][reg].op = \text{"Free"}]$  The register is freed.

$\wedge \text{LET } idx \triangleq \text{CHOOSE } i \in \text{DOMAIN } opQ[proc] : opQ[proc][i].reg = reg$  The appropriate  $opQ$  entry is updated.

$\text{IN } opQ' = [opQ \text{ EXCEPT } ![proc][idx].reg = Done]$

$\wedge UpdateOpOrder$   $opOrder$  is updated.

**Figure 11.6b:** Module *InnerSerial* (middle).

$RespondToRd(proc, reg) \triangleq$  The memory responds to a read request in processor  $proc$ 's register  $reg$ .  
 LET  $req \triangleq regFile[proc][reg]$   $proc$ 's register  $reg$  contains the request  $req$ , which is in  $opQ[proc][idx]$ .  
 $idx \triangleq \text{CHOOSE } i \in \text{DOMAIN } opQ[proc] : opQ[proc][i].reg = reg$   
 IN  $\wedge req.op = \text{"Rd"}$  The register must contain an active read request.  
 $\wedge \exists src \in \text{goodSource}([proc \mapsto proc, idx \mapsto idx]) :$  The read obtains its value from a source  $src$ .  
 LET  $val \triangleq \text{IF } src = \text{InitWr} \text{ THEN } InitMem[req.adr] \text{ ELSE } opIdQ(src).req.val$  The value returned by the read.  
 IN  $\wedge regFile' = [regFile \text{ EXCEPT } ![proc][reg].val = val, ![proc][reg].op = \text{"Free"}]$  Set register's  $val$  field, and free the register.  
 $\wedge opQ' = [opQ \text{ EXCEPT } opQ[proc][idx] \text{ is updated appropriately.} ![proc][idx] = [req \mapsto [req \text{ EXCEPT } !.val = val], reg \mapsto Done, source \mapsto src]]$   
 $\wedge UpdateOpOrder$   $opOrder$  is updated.  
 $Internal \triangleq \wedge \text{UNCHANGED } \langle regFile, opQ \rangle \wedge UpdateOpOrder$   
 $Next \triangleq$  The next-state action.  
 $\vee \exists proc \in Proc, reg \in Reg : \vee \exists req \in Request : IssueRequest(proc, req, reg)$   
 $\vee RespondToRd(proc, reg)$   
 $\vee RespondToWr(proc, reg)$   
 $\vee Internal$

---

$Spec \triangleq$  The complete internal specification.  
 $\wedge Init$   
 $\wedge \square[Next]_{\langle regFile, opQ, opOrder \rangle}$   
 $\wedge \forall proc \in Proc, reg \in Reg :$  The memory eventually responds to every request.  
 $\text{WF}_{\langle regFile, opQ, opOrder \rangle}(\text{RespondToWr}(proc, reg) \vee \text{RespondToRd}(proc, reg))$   
 $\wedge \forall oi, oj \in [proc : Proc, idx : Nat] :$  All operations are eventually ordered.  
 $(oi \neq oj) \Rightarrow \text{WF}_{\langle regFile, opQ, opOrder \rangle}(\wedge (oi \in opId) \wedge (oj \in opId)$   
 $\wedge Internal$   
 $\wedge ((oi, oj) \in opOrder') \vee ((oj, oi) \in opOrder'))$

---

THEOREM  $Spec \Rightarrow \square(\text{DataInvariant} \wedge \text{Serializable})$

Figure 11.6c: Module *InnerSerial* (end).

*RespondToRd*, *RespondToWr*, and *Internal*. Like the serial memory specification, it has an internal variable  $opQ$  to which the *IssueRequest* operation appends an entry with *req* (request) and *reg* (register) fields. However, an operation does not remain forever in  $opQ$ . Instead, an *Internal* step removes it after it has been completed. The specification has a second internal variable *mem* that represents the contents of a memory—that is, the value of *mem* is a function from *Adr* to *Val*. The value of *mem* is changed only by an *Internal* action that removes a write from  $opQ$ .

Recall that the correctness condition has two requirements:

1. There is a sequential execution order of all the operations that explains the values returned by reads.
2. This execution order is consistent with the order in which operations are issued by each individual processor.

The order in which operations are removed from  $opQ$  is an explanatory execution order that satisfies requirement 1 if the *Internal* action satisfies these properties:

- When a write of value *val* to address *adr* is removed from  $opQ$ , the value of  $mem[adr]$  is set to *val*.
- A read of address *adr* that returned a value *val* can be removed from  $opQ$  only if  $mem[adr] = val$ .

Requirement 2 is satisfied if operations issued by processor *p* are appended by the *IssueRequest* action to the tail of  $opQ[p]$ , and are removed by the *Internal* action only from the head of  $opQ[p]$ .

We have now determined what the *IssueRequest* and *Internal* actions should do. The *RespondToWr* action is obvious; it's essentially the same as in the serial memory specification. The problem is the *RespondToRd* action. How can we define it so that the value returned by a read is one that *mem* will contain when the *Internal* action has to remove the read from  $opQ$ ? The answer is surprisingly simple: we allow the read to return any value. If the read were to return a bad value—for example, one that is never written—then the *Internal* action would never be able to remove the read from  $opQ$ . We rule out that possibility with a liveness condition requiring that every operation in  $opQ$  eventually be removed. This makes it easy to write the *Internal* action. The only remaining problem is expressing the liveness condition.

To guarantee that every operation is eventually removed from  $opQ$ , it suffices to guarantee that, for every processor *proc*, the operation at the head of  $opQ[proc]$  is eventually removed. The desired liveness condition can therefore be expressed as

$$\forall proc \in Proc : WF_{\langle \dots \rangle}(RemoveOp(proc))$$

where  $\text{RemoveOp}(\text{proc})$  is an action that unconditionally removes the operation from the head of  $\text{opQ}[\text{proc}]$ . For convenience, we let the  $\text{RemoveOp}(\text{proc})$  action also update  $\text{mem}$ . We then define a separate action  $\text{Internal}(\text{proc})$  for each processor  $\text{proc}$ . It conjoins to  $\text{RemoveOp}(\text{proc})$  the following enabling condition, which asserts that if the operation being removed is a read, then it has returned the correct value:

$$\begin{aligned} (\text{Head}(\text{opQ}[\text{proc}]).\text{req}.op = \text{"Rd"}) \Rightarrow \\ (\text{mem}[\text{Head}(\text{opQ}[\text{proc}]).\text{req}.adr] = \text{Head}(\text{opQ}[\text{proc}]).\text{req}.val) \end{aligned}$$

The complete internal specification, with the variables  $\text{opQ}$  and  $\text{mem}$  visible, appears in module *InnerSequential* on the following two pages. At this point, you should have no trouble understanding it. You should also have no trouble writing a module that instantiates *InnerSequential* and hides the internal variables  $\text{opQ}$  and  $\text{mem}$  to produce the final specification, so I won't bother doing it for you.

### 11.2.5 The Memory Specifications Considered

Almost every specification we write admits a direct implementation, based on the initial predicate and next-state action. Such an implementation may be completely impractical, but it is theoretically possible. It's easy to implement the linearizable memory with a single central memory. A direct implementation of the serial memory would require maintaining queues of all operations issued thus far, and a computationally infeasible search for possible total orderings. But, in theory, it's easy.

Our specification of a sequentially consistent memory cannot be implemented directly. A direct implementation would have to guess the correct value to return on a read, which is impossible. The specification is not directly implementable because it is not machine closed. As explained in Section 8.9.2 on page 111, a non-machine-closed specification is one in which a direct implementation can “paint itself into a corner,” reaching a point at which it is no longer possible to satisfy the specification. Any finite scenario of memory operations can be produced by a behavior satisfying the sequentially consistent memory's initial predicate and next-state action—namely, a behavior that contains no *Internal* steps. However, not every finite scenario can be extended to one that is explainable by a sequential execution. For example, no scenario that begins as follows is possible in a two-processor system:

$$\begin{aligned} \text{Processor } p: \text{Wr}_p(a1, v1), \text{ Rd}_p(a1, v2), \text{ Wr}_p(a2, v2) \\ \text{Processor } q: \text{Wr}_q(a2, v1), \text{ Rd}_q(a2, v2), \text{ Wr}_q(a1, v2) \end{aligned}$$

Here's why:

This notation for describing scenarios was introduced on page 186.

MODULE *InnerSequential*EXTENDS *RegisterInterface*, *Naturals*, *Sequences*, *FiniteSets*VARIABLE  $opQ$ ,  $opQ[p][i]$  is the  $i^{\text{th}}$  operation issued by processor  $p$ . $mem$  An internal memory. $Done \triangleq \text{CHOOSE } v : v \notin Reg \quad \text{The } reg \text{ field value for a completed operation.}$  $DataInvariant \triangleq$  $\wedge RegFileTypeInvariant$ Simple type invariants for  $regFile$ ,  
 $opQ$ , and  
 $mem$ . $\wedge opQ \in [Proc \rightarrow Seq([req : Request, reg : Reg \cup \{Done\}])]$  $\wedge mem \in [Adr \rightarrow Val]$  $\wedge \forall p \in Proc, r \in Reg : \quad \text{Only nonfree registers have corresponding } opQ \text{ entries.}$  $Cardinality(\{i \in \text{DOMAIN } opQ[p] : opQ[p][i].reg = r\}) =$ IF  $regFile[p][r].op = \text{``Free''}$  THEN 0 ELSE 1 $Init \triangleq \text{The initial predicate.}$  $\wedge regFile \in [Proc \rightarrow [Reg \rightarrow FreeRegValue]] \quad \text{Every register is free.}$  $\wedge opQ = [p \in Proc \mapsto \langle \rangle] \quad \text{There are no operations in } opQ.$  $\wedge mem \in [Adr \rightarrow Val] \quad \text{The internal memory can have any initial contents.}$  $IssueRequest(proc, req, reg) \triangleq \text{Processor } proc \text{ issues request } req \text{ in register } reg.$  $\wedge regFile[proc][reg].op = \text{``Free''} \quad \text{The register must be free.}$  $\wedge regFile' = [regFile \text{ EXCEPT } !(proc)[reg] = req] \quad \text{Put request in register.}$  $\wedge opQ' = [opQ \text{ EXCEPT } !(proc) = Append(@, [req \mapsto req, reg \mapsto reg])] \quad \text{Add request to } opQ[proc].$  $\wedge \text{UNCHANGED } mem$  $RespondToRd(proc, reg) \triangleq \text{The memory responds to a read request in processor } proc \text{'s register } reg.$  $\wedge regFile[proc][reg].op = \text{``Rd''} \quad \text{The register must contain an active read request.}$  $\wedge \exists val \in Val : \quad val \text{ is the value returned.}$  $\wedge regFile' = [regFile \text{ EXCEPT } !(proc)[reg].val = val, \quad \text{Set the register's } val \text{ field,} \\ !(proc)[reg].op = \text{``Free''}] \quad \text{and free the register.}$  $\wedge opQ' = \text{LET } idx \triangleq opQ[proc][idx] \text{ contains the request in register } reg.$ CHOOSE  $i \in \text{DOMAIN } opQ[proc] : opQ[proc][i].reg = reg$ IN  $[opQ \text{ EXCEPT } !(proc)[idx].req.val = val, \quad \text{Set } opQ[proc][idx] \text{'s } val \text{ field to} \\ !(proc)[idx].reg = Done] \quad val \text{ and its } reg \text{ field to } Done.$  $\wedge \text{UNCHANGED } mem$ **Figure 11.7a:** Module *InnerSequential* (beginning).

$RespondToWr(proc, reg) \triangleq$  The memory responds to a write request in processor  $proc$ 's register  $reg$ .  
 $\wedge regFile[proc][reg].op = "Wr"$  The register must contain an active write request.  
 $\wedge regFile' = [regFile \text{ EXCEPT } !(proc)[reg].op = "Free"]$  Free the register.  
 $\wedge \text{LET } idx \triangleq \text{CHOOSE } i \in \text{DOMAIN } opQ[proc] : opQ[proc][i].reg = reg$   
 $\text{IN } opQ' = [opQ \text{ EXCEPT } !(proc)[idx].reg = Done]$  Update the appropriate  $opQ$  entry.  
 $\wedge \text{UNCHANGED } mem$

$RemoveOp(proc) \triangleq$  Unconditionally remove the operation at the head of  $opQ[proc]$  and update  $mem$ .  
 $\wedge opQ[proc] \neq \langle \rangle$   $opQ[proc]$  must be nonempty.  
 $\wedge Head(opQ[proc]).reg = Done$  The operation must have been completed.  
 $\wedge mem' = \text{IF } Head(opQ[proc]).req.op = "Rd"$  Leave  $mem$  unchanged for a  
 $\text{THEN } mem$  read operation, update it for  
 $\text{ELSE } [mem \text{ EXCEPT } !(Head(opQ[proc]).req.adr) = Head(opQ[proc]).req.val]$  a write operation.  
 $\wedge opQ' = [opQ \text{ EXCEPT } !(proc) = Tail(@)]$  Remove the operation from  $opQ[proc]$ .  
 $\wedge \text{UNCHANGED } regFile$  No register is changed.

$Internal(proc) \triangleq$  Remove the operation at the head of  $opQ[proc]$ . But if it's a read,  
 $\wedge RemoveOp(proc)$  only do so if it returned the value now in  $mem$ .  
 $\wedge (Head(opQ[proc]).req.op = "Rd") \Rightarrow$   
 $(mem[Head(opQ[proc]).req.adr] = Head(opQ[proc]).req.val)$

$Next \triangleq$  The next-state action.  
 $\exists proc \in Proc : \vee \exists reg \in Reg : \vee \exists req \in Request : IssueRequest(proc, req, reg)$   
 $\quad \vee RespondToRd(proc, reg)$   
 $\quad \vee RespondToWr(proc, reg)$   
 $\quad \vee Internal(proc)$

---

$Spec \triangleq \wedge Init$   
 $\wedge \square[Next](regFile, opQ, mem)$   
 $\wedge \forall proc \in Proc, reg \in Reg : \quad \text{The memory eventually responds to every request.}$   
 $\quad \text{WF}_{(regFile, opQ, mem)}(RespondToWr(proc, reg) \vee RespondToRd(proc, reg))$   
 $\wedge \forall proc \in Proc : \quad \text{Every operation is eventually removed from } opQ.$   
 $\quad \text{WF}_{(regFile, opQ, mem)}(RemoveOp(proc))$

---

THEOREM  $Spec \Rightarrow \square DataInvariant$

Figure 11.7b: Module *InnerSequential* (end).

$Wr_q(a1, v2)$

|                         |                                                         |
|-------------------------|---------------------------------------------------------|
| precedes $Rd_p(a1, v2)$ | This is the only explanation of the value read by $p$ . |
| precedes $Wr_p(a2, v2)$ | By the order in which operations are issued.            |
| precedes $Rd_q(a2, v2)$ | This is the only explanation of the value read by $q$ . |
| precedes $Wr_q(a1, v2)$ | By the order in which operations are issued.            |

Hence  $q$ 's write of  $a1$  must precede itself, which is impossible.

As mentioned in Section 8.9.2, a specification is machine closed if the liveness property is the conjunction of fairness properties for actions that imply the next-state action. The sequential memory specification asserts weak fairness of  $RemoveOp(proc)$ , for processors  $proc$ , and  $RemoveOp(proc)$  does not imply the next-state action. (The next-state action does not allow a  $RemoveOp(proc)$  step that removes from  $opQ[proc]$  a read that has returned the wrong value.)

Very high-level system specifications, such as our memory specifications, are subtle. It's easy to get them wrong. The approach we used in the serial memory specification—namely, writing conditions on the history of all operations—is dangerous. It's easy to forget some conditions. A non-machine-closed specification can occasionally be the simplest way to express what you want so say.



# Part III

# The Tools



## Chapter 12

# The Syntactic Analyzer

The Syntactic Analyzer is a Java program, written by Jean-Charles Grégoire and David Jefferson, that parses a TLA<sup>+</sup> specification and checks it for errors. The analyzer also serves as a front end for other tools, such as TLC (see Chapter 14). It is available from the TLA Web page.

You will probably run the analyzer by typing the command

*program-name* *option* *spec-file*

where

*program-name* depends on your particular system. It might be

`java tlasany.SANY`

*spec-file* is the name of the file containing the TLA<sup>+</sup> specification. Each module named *M* that appears in the specification (except for submodules) must be in a separate file named *M.tla*. The extension *.tla* may be omitted from *spec-file*.

*option* is either empty or consists of one of the following two options:

- s Causes the analyzer to check only for syntactic errors and not for semantic errors. (These two classes of error are explained below.) You can use this option to find syntax errors when you begin writing a specification.
- d Causes the analyzer to enter debugging mode after checking the specification. In this mode, you can examine the specification's structure—for example, finding out where it thinks a particular identifier is defined or declared. The documentation that comes with the analyzer explains how to do this.

The rest of this brief chapter provides some hints for what to do when the Syntactic Analyzer reports an error.

The errors that the analyzer detects fall into two separate classes, which are usually called *syntactic* and *semantic* errors. A syntactic error is one that makes the specification grammatically incorrect, meaning that it violates either the BNF grammar or the precedence and alignment rules, described in Chapter 15. A semantic error is one that violates the legality conditions mentioned in Chapter 17. The term *semantic error* is misleading, because it suggests an error that makes a specification have the wrong meaning. All errors found by the analyzer are ones that make the specification illegal—that is, not syntactically well-formed—and hence make it have no meaning at all.

The analyzer reads the file sequentially, starting from the beginning, and it reports a syntax error if and when it reaches a point at which it becomes impossible for any continuation to produce a grammatically correct specification. For example, if we omitted the colon after  $\exists req \in MReq$  in the definition of *Req* from module *InternalMemory* on page 52, we would get

$$\begin{aligned} Req(p) \triangleq & \ \wedge \ ctl[p] = "rdy" \\ & \wedge \exists req \in MReq \wedge Send(p, req, memInt, memInt') \\ & \quad \wedge buf' = [buf \text{ EXCEPT } !(p) = req] \\ & \quad \wedge ctl' = [ctl \text{ EXCEPT } !(p) = "busy"] \\ & \wedge \text{UNCHANGED } mem \end{aligned}$$

This would cause the analyzer to print something like

```
Parse Error
Encountered "/" at line 19, column 11
```

Line 19, column 11 is the position of the  $\wedge$  that begins the last line of the definition (right before the UNCHANGED). Until then, the analyzer thought it was parsing a quantified expression that began

$$\exists req \in (MReq \wedge Send(p, req, memInt, memInt') \wedge buf' = \dots$$

(Such an expression is silly, having the form  $\exists req \in p : \dots$  where  $p$  is a Boolean, but it's legal.) The analyzer was interpreting each of these  $\wedge$  symbols as an infix operator. However, interpreting the last  $\wedge$  of this definition (at line 19, column 11) as an infix operator would violate the alignment rules for the outer conjunction list, so the analyzer reported an error.

As this example suggests, the analyzer may discover a syntax error far past the actual mistake. To help you locate the problem, it prints out a trace of where it was in the parse tree when it found the error. For this example, it prints

---

Residual stack trace follows:

Quantified form starting at line 16, column 14.

Junction Item starting at line 16, column 11.

AND-OR Junction starting at line 15, column 11.

Definition starting at line 15, column 1.

Module body starting at line 3, column 1.

If you can't find the source of an error, try the "divide and conquer" method: keep removing different parts of the module until you isolate the source of the problem.

Semantic errors are usually easy to find because the analyzer can locate them precisely. A typical semantic error is an undefined symbol that arises because you mistype an identifier. If, instead of leaving out the colon in the definition of *Req(p)*, we had left out the *e* in *MReq*, the analyzer would have reported

```
line 16, col 26 to line 16, col 28 of module InternalMemory
```

```
Could not resolve name 'MReq'.
```

The analyzer stops when it encounters the first syntactic error. It can detect multiple semantic errors in a single run.



# Chapter 13

## The TLATEX Typesetter

TLATEX is a Java program for typesetting TLA<sup>+</sup> modules, based on ideas by Dmitri Samborski. It can be obtained through the TLA Web page.

### 13.1 Introduction

TLATEX calls the LATEX program to do the actual typesetting. LATEX is a document-production system based on Donald Knuth's TEX typesetting program.<sup>1</sup> LATEX normally produces as its output a *dvi file*—a file with extension *dvi* containing a device-independent description of the typeset output. TLATEX has options that allow it to call another program to translate the *dvi* file into a PostScript or PDF file. Some versions of LATEX produce a PDF file directly.

You must have LATEX installed on your computer to run TLATEX. LATEX is public-domain software that can be downloaded from the Web; proprietary versions are also available. The TLA Web page points to the TLATEX Web page, which contains information about obtaining LATEX and a PostScript or PDF converter.

You will probably run TLATEX by typing

```
java tlatex.TLA [options] fileName
```

where *fileName* is the name of the input file, and [*options*] is an optional sequence of options, each option name preceded by “-”. Some options are followed by an argument, a multi-word argument being enclosed in double-quotes. If *fileName* does not contain an extension, then the input file is *fileName.tla*. For example, the command

---

<sup>1</sup>LATEX is described in *LATEX: A Document Preparation System, Second Edition*, by Leslie Lamport, published by Addison-Wesley, Reading, Massachusetts, 1994. TEX is described in *The T<sub>E</sub>Xbook* by Donald E. Knuth, published by Addison-Wesley, Reading, Massachusetts, 1986.

---

```
java tlatex.TLA -ptSize 12 -shade MySpec
```

typesets the module in the file *MySpec.tla* using the *ptSize* option with argument 12 and the *shade* option. The input file must contain a complete TLA<sup>+</sup> module. Running TLATEX with the *help* option produces a list of all options. Running it with the *info* option produces most of the information contained in this chapter. (The *fileName* argument can be omitted when using the *help* or *info* option.)

All you probably need to know about using TLATEX is

- TLATEX can shade comments, as explained in the next section.
- The next section also explains how to get TLATEX to produce a PostScript or PDF file.
- The *number* option causes TLATEX to print line numbers in the left margin.
- You should use the *latexCommand* option if you run LATEX on your system by typing something other than *latex*. For example, if you run LATEX on file *f.tex* by typing

```
locallatex f.tex
```

then you should run TLATEX by typing something like

```
java tlatex.TLA -latexCommand locallatex fileName
```

- If you happen to use any of these three two-character sequences in a comment:

```
“~” “^” “`”
```

then you'd better read Section 13.4 on page 214 to learn about how TLATEX formats comments.

TLATEX's output should be good enough for most purposes. The following sections describe how you can get TLATEX to do a better job, and what to do in the unlikely case that it produces weird output.

## 13.2 Comment Shading

The *shade* option causes TLATEX to typeset comments in shaded boxes. A specification generally looks best when comments are shaded, as they are in this book. However, shading is not supported by some programs for viewing and printing dvi files. Hence, it may be necessary to create a PostScript or PDF file from the dvi file to view a specification with shaded comments. Here are all the options relevant to shading.

**-grayLevel** *num*

Determines the darkness of the shading, where *num* is a number between 0 and 1. The value 0 means completely black, and 1 means white; the default value is .85. The actual degree of shading depends on the output device and can vary from printer to printer and from screen to screen. You will have to experiment to find the right value for your system.

**-ps****-nops**

These options tell TLATEX to create or not to create a PostScript or PDF output file. The default is to create one if the *shade* option is specified, and otherwise not to.

**-psCommand** *cmd*

This is the command run by TLATEX to produce the PostScript or PDF output file. Its default value is dvips. TLATEX calls the operating system with the command

*cmd* *dviFile*

where *dviFile* is the name of the dvi file produced by running LATEX. If a more sophisticated command is needed, you may want to use the *nops* option and run a separate program to create the PostScript or PDF file.

### 13.3 How It Typesets the Specification

TLATEX should typeset the specification itself pretty much the way you would want it to. It preserves most of the meaningful alignments in the specification—for example:

| <b>Input</b>            | <b>Output</b>                           |
|-------------------------|-----------------------------------------|
| Action == /\ x' = x - y | Action $\triangleq$ $\wedge x' = x - y$ |
| /\ yy' = 123            | $\wedge yy' = 123$                      |
| /\ zzz' = zzz           | $\wedge zzz' = zzz$                     |

Observe how the  $\wedge$  and  $=$  symbols are aligned in the output. Extra spaces in the input will be reflected in the output. However, TLATEX treats no space and one space between symbols the same:

| <b>Input</b> | <b>Output</b> |
|--------------|---------------|
| x+y          | $x + y$       |
| x + y        | $x + y$       |
| x + y        | $x + y$       |

TLATEX typesets the single TLA<sup>+</sup> module that must appear in the input file. It will also typeset any material that precedes and follows the module as if

it were a comment. (However, that text won't be shaded.) The *noProlog* and *noEpilog* options suppress typesetting of material that precedes and follows the module, respectively.

TLATEX does not check that the specification is syntactically correct TLA<sup>+</sup> input. However, it will report an error if the specification contains an illegal lexeme, such as “;”.

## 13.4 How It Typesets Comments

TLATEX distinguishes between one-line and multi-line comments. A one-line comment is any comment that is not a multi-line comment. Multi-line comments can be typed in any of the following three styles:

|                                                        |                                               |                                |
|--------------------------------------------------------|-----------------------------------------------|--------------------------------|
| *****<br>(* This is    *)<br>(* a comment. *)<br>***** | *****<br>\* This is<br>\* a comment.<br>***** | (* This<br>is a<br>comment. *) |
|--------------------------------------------------------|-----------------------------------------------|--------------------------------|

In the first two styles, the `(*` or `\*` characters on the left must all be aligned, and the last line (containing the comment `*****`) is optional. In the first style, nothing may appear to the right of the comment—otherwise, the input is considered to be a sequence of separate one-line comments. The third style works best when nothing appears on the same line to the left of the `(*` or to the right of the `*)`.

TLATEX tries to do a sensible job of typesetting comments. In a multi-line comment, it usually considers a sequence of non-blank lines to be a single paragraph, in which case it typesets them as one paragraph and ignores line breaks in the input. But it does try to recognize tables and other kinds of multi-line formatting when deciding where to break lines. You can help it as follows:

- End each sentence with a period (“.”).
- Add blank lines to indicate the logical separation of items.
- Left-align the lines of each paragraph.

Below are some common ways in which TLATEX can mess up the typesetting of comments, and what you can do about it.

TLATEX can confuse parts of a specification with ordinary text. For example, identifiers should be italicized, and the minus in the expression  $x - y$  should be typeset differently from the dash in *x-ray*. TLATEX gets this right most of the time, but it does make mistakes. You can tell TLATEX to treat something as part of a specification by putting single quotes (‘ and ’) around it. You can tell it to treat something as ordinary text by putting ‘^ and ^’ around it. For example:

| Input                                                                             | Output                                                    |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------|
| \*****<br>/* A better value of 'bar' is<br>/* now in '^http://foo/bar'.<br>\***** | A better value of <i>bar</i> is now in<br>http://foo/bar. |

But this is seldom necessary; TLATEX usually does the right thing.

**Warning:** Do not put any character between ‘^’ and ‘^’ except letters, numbers, and ordinary punctuation—unless you know what you’re doing. In particular, the following characters have special meaning to LATEX and can have unexpected effects if used between ‘^’ and ‘^’:

\_ ~ # \$ % ^ & < > \ " | { }

See Section 13.8 on page 219 for further information about what can go between ‘^’ and ‘^’.

TLATEX isn’t very good at copying the way paragraphs are formatted in a comment. For example, note how it fails to align the two *A*s in

| Input                   | Output               |
|-------------------------|----------------------|
| \*****                  |                      |
| /* gnat: A tiny insect. | gnat: A tiny insect. |
| /*                      |                      |
| /* gnu: A short word.   | gnu: A short word.   |
| \*****                  |                      |

You can tell TLATEX to typeset a sequence of lines precisely the way they appear in the input, using a fixed-width font, by enclosing the lines with ‘. and ‘.’, as in

| Input                    | Output                |
|--------------------------|-----------------------|
| \*****                   |                       |
| /* This explains it all: | This explains it all: |
| /*                       | ---                   |
| /* . --- ---             | P  --->  M            |
| /*   P  --->  M          | ---                   |
| /* --- --- . ,           | ---                   |
| \*****                   |                       |

Using ‘. and ‘.’ is the only reasonable thing to do for a diagram. However, if you know (or want to learn) LATEX, Section 13.8 below on using LATEX commands in comments will explain how you can get TLATEX to do a good job of formatting things like lists and tables.

TLATEX will occasionally typeset a paragraph very loosely, with one or more lines containing lots of space between the words. This happens if there is no good way to typeset the paragraph. If it bothers you, the easiest solution is to rewrite

the paragraph. You can also try to fix the problem with L<sup>A</sup>T<sub>E</sub>X commands. (See Section 13.8 below.)

TLATEX usually handles pairs of double-quote characters ("") the way it should:

**Input**

```

* The string "ok" is
* a "good" value.

```

**Output**

The string “ok” is a “good” value.

However, if it gets confused, you can use single quotes to identify string values and ‘‘ and ’’ to produce the left and right double-quotes of ordinary text:

**Input**

```

* He asks ‘‘Is “good”,
* bad?’’

```

**Output**

He asks “Is “good” bad?”

TLATEX ignores any (\* ... \*) comment that appears within another comment. So, you can get it not to typeset part of a comment by enclosing that part between (\* and \*). But a better way to omit part of a comment is to enclose it between ‘~ and ~’:

**Input**

```

* x+y is always ‘~I
* hope~’ positive.

```

**Output**

$x + y$  is always positive.

## 13.5 Adjusting the Output Format

The following options allow you to adjust the font size, the dimensions of the printed area, and the position of the text on the page:

**-ptSize num**

Specifies the size of the font. Legal values of *num* are 10, 11, or 12, which cause the specification to be typeset in a 10-, 11-, or 12-point font. The default value is 10.

**-textwidth num**

**-textheight num**

The value of *num* specifies the width and height of the typeset output, in points. A point is 1/72 of an inch, or about 1/3 mm.

**-hoffset** *num*  
**-voffset** *num*

The value of *num* specifies the distance, in points, by which the text should be moved horizontally or vertically on the page. Exactly where on a page the text appears depends on the printer or screen-display program. You may have to adjust this value to get the output to appear centered on the printed page, or for the entire output to be visible when viewed on the screen.

## 13.6 Output Files

TLATEX itself writes either two or three files, depending on the options. The names of these files are normally determined from the name of the input file. However, options allow you to specify the name of each of these files. TLATEX also runs the separate LATEX program and possibly a program to produce a PostScript or PDF file. These programs produce additional files. Below are the file-related options. In their descriptions, the root of a file name is the name with any extension or path specifier removed; for example, the root of *c:\foo\bar.tla* is *bar*. All file names are interpreted relative to the directory in which TLATEX is run.

**-out** *fileName*

If *f* is the root of *fileName*, then *f.tex* is the name of the LATEX input file that TLATEX writes to produce the final output. TLATEX then runs LATEX with *f.tex* as input, producing the following files:

*f.dvi* The dvi output file.

*f.log* A log file, containing LATEX's messages. In this file, an *overfull hbox* warning means that a specification line is too wide and extends into the right margin, and an *underfull hbox* warning means that LATEX could find no good line breaks in a comment paragraph. Unfortunately, the line numbers in the file refer to the *f.tex* file, not to the specification. But by examining the *f.tex* file, you can probably figure out where the corresponding part of the specification is.

*f.aux* A LATEX auxiliary file that is of no interest.

The default *out* file name is the root of the input file name.

**-alignOut** *fileName*

This specifies the root name of the LATEX alignment file TLATEX writes—a file described in Section 13.7 below on trouble-shooting. If *f* is the root of *fileName*, then the alignment file is named *f.tex*, and running LATEX on it produces the files *f.dvi*, *f.log*, and *f.aux*. Only the *f.log* file is of interest. If

the *alignOut* option is not specified, the alignment file is given the same name as the *out* file. This option is used only for trouble-shooting, as described in the section below.

**-tlaOut** *fileName*

This option causes TLATEX to write to *fileName* a file that is almost the same as the input file. (The extension *tla* is added to *fileName* if it has no extension.) The *tlaOut* file differs from the input in that any portion of a comment enclosed by ‘^ and ^’ is removed, and every occurrence of the two-character strings

‘~’ ‘~’ ‘. . .’

is replaced by two blanks. As explained in Section 13.8 below, the *tlaOut* option allows you to maintain a version of the specification that is readable in ASCII while using LATEX commands to provide high-quality typesetting of comments. The default is not to write a *tlaOut* file.

**-style** *fileName*

This option is of interest only to LATEX users. Normally, TLATEX inserts a copy of the *tlatex* package file in the LATEX input files that it writes. The *style* option causes it instead to insert a `\usepackage` command to read the LATEX package named *fileName*. (LATEX package files have the extension *sty*. That extension is added to *fileName* if it’s not already there.) The TLATEX style defines a number of special commands that are written by TLATEX in its LATEX input files. The package file specified by the *style* option must also define those commands. Any package file should therefore be created by modifying the standard *tlatex* package, which is the file *tlatex.sty* in the same directory as TLATEX’s Java program files. You might want to create a new package to change the way TLATEX formats the specification, or to define additional commands for use in ‘^...^’ text in comments.

## 13.7 Trouble-Shooting

TLATEX’s error messages should be self-explanatory. However, it calls upon the operating system up to three times to execute other programs:

- It runs LATEX on the *alignOut* file that it wrote.
- It runs LATEX on the *out* file that it wrote.
- It may execute the *psCommand* to create the PostScript or PDF output file.

After each of the last two executions, TLATEX writes a message asserting that the appropriate output file was written. It might lie. Any of those executions might fail, possibly causing no output file to be written. Such a failure can even cause the operating system never to return control to TLATEX, so TLATEX never terminates. This type of failure is the likely problem if TLATEX does not produce a dvi file or a PostScript/PDF file, or if it never terminates. In that case, you should try rerunning TLATEX using the *alignOut* option to write a separate alignment file. Reading the two log files that LATEX produces, or any error file produced by executing *psCommand*, may shed light on the problem.

Normally, the LATEX input files written by TLATEX should not produce any LATEX errors. However, incorrect LATEX commands introduced in ‘^...^’ regions can cause LATEX to fail.

## 13.8 Using LATEX Commands

TLATEX puts any text enclosed between ‘^ and ^’ in a comment into the LATEX input file exactly as it appears. This allows you to insert LATEX formatting commands in comments. There are two ways to use this.

- You can enclose between ‘^ and ^’ a short phrase appearing on a single line of input. LATEX typesets that phrase as part of the enclosing paragraph.
- You can enclose one or more complete lines of a multi-line comment between ‘^ and ^’. That text is typeset as one or more separate paragraphs whose prevailing left margin is determined by the position of the ‘^’, as shown here:

| Input                   | Output                   |
|-------------------------|--------------------------|
| \*****                  |                          |
| \* The first paragraph. | The first paragraph.     |
| \*                      |                          |
| \* The 2nd paragraph.   | The 2nd paragraph.       |
| \*                      |                          |
| \* ^ Text formatted     |                          |
| \* by \LaTeX. ^         | Text formatted by LATEX. |
| \*****                  |                          |

LATEX typesets the text between ‘^ and ^’ in LR mode for a one-line comment and in paragraph mode for a multi-line comment. The LATEX file produced by TLATEX defines a **describe** environment that is useful for formatting text in a multi-line ‘^...^’ region. This environment is the same as the standard LATEX **description** environment, except that it takes an argument, which should be the widest item label in the environment:

| Input                         | Output             |
|-------------------------------|--------------------|
| \*****                        |                    |
| \* `^ \begin{describe}{gnat:} | gnat: Tiny insect. |
| \* \item[gnat:] Tiny insect.  |                    |
| \* \item[gnu:] Short word.    | gnu: Short word.   |
| \* \end{describe} ^,          |                    |
| \*****                        |                    |

As this example shows, putting L<sup>A</sup>T<sub>E</sub>X commands in comments makes the comments in the input file rather unreadable. You can maintain both a typeset and an ASCII-readable version of the specification by enclosing text that should appear only in the ASCII version between `^ and ^'. You can then accompany each `^...^' region with its ASCII version enclosed by `^ and ^'. For example, the input file could contain

```

* `^ \begin{describe}{gnat:}
* \item[gnat:] A tiny insect.
* \item[gnu:] A short word.
* \end{describe} ^
* `^ gnat: A tiny insect.
*
* gnu: A short word. ^,

```

The *tlaOut* option causes TLATEX to write a version of the original specification with `^...^' regions deleted, and with `^' and `^' strings replaced by spaces. (The strings `.' and `.' are also replaced by spaces.) In the example above, the *tlaOut* file would contain the comment

```

*
* gnat: A tiny insect.
*
* gnu: A short word.

```

The blank line at the top was produced by the end-of-line character that follows the ^'.

**Warning:** An error in a L<sup>A</sup>T<sub>E</sub>X command inside `^...^' text can cause TLATEX not to produce any output. See Section 13.7 above on trouble-shooting.

# Chapter 14

## The TLC Model Checker

TLC is a program for finding errors in TLA<sup>+</sup> specifications. It was designed and implemented by Yuan Yu, with help from Leslie Lamport, Mark Hayden, and Mark Tuttle. It is available through the TLA Web page. This chapter describes TLC Version 2. At the time I am writing this, Version 2 is still being implemented and only Version 1 is available. Consult the documentation that accompanies the software to find out what version it is and how it differs from the version described here.

### 14.1 Introduction to TLC

TLC handles specifications that have the standard form

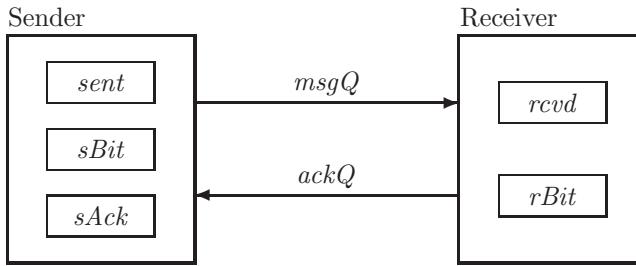
$$(14.1) \quad \text{Init} \wedge \square[\text{Next}]_{\text{vars}} \wedge \text{Temporal}$$

where *Init* is the initial predicate, *Next* is the next-state action, *vars* is the tuple of all variables, and *Temporal* is a temporal formula that usually specifies a liveness condition. Liveness and temporal formulas are explained in Chapter 8. If your specification contains no *Temporal* formula, so it has the form  $\text{Init} \wedge \square[\text{Next}]_{\text{vars}}$ , then you can ignore the discussion of temporal checking. TLC does not handle the hiding operator  $\exists$  (temporal existential quantification). You can check a specification with hidden variables by checking the internal specification, in which those variables are visible.

The most effective way to find errors in a specification is by trying to verify that it satisfies properties that it should. TLC can check that the specification satisfies (implies) a large class of TLA formulas—a class whose main restriction is that formulas may not contain  $\exists$ . You can also run TLC without having it check any property, in which case it will just look for two kinds of errors:

- “Silliness” errors. As explained in Section 6.2, a silly expression is one like  $3 + \langle 1, 2 \rangle$ , whose meaning is not determined by the semantics of TLA<sup>+</sup>. A specification is incorrect if whether or not some particular behavior satisfies it depends on the meaning of a silly expression.
- Deadlock. The absence of deadlock is a particular property that we often want a specification to satisfy; it is expressed by the invariance property  $\square(\text{ENABLED } \text{Next})$ . A counterexample to this property is a behavior exhibiting deadlock—that is, reaching a state in which *Next* is not enabled, so no further (nonstuttering) step is possible. TLC normally checks for deadlock, but this checking can be disabled since, for some systems, deadlock may just indicate successful termination.

The use of TLC will be illustrated with a simple example—a specification of the alternating bit protocol for sending data over a lossy FIFO transmission line. An algorithm designer might describe the protocol as a system that looks like this:



The sender can send a value when the one-bit values *sBit* and *sAck* are equal. It sets the variables *sent* to the value it is sending and complements *sBit*. This value is eventually delivered to the receiver by setting the variable *rcvd* and complementing the one-bit value *rBit*. Some time later, the sender’s *sAck* value is complemented, permitting the next value to be sent. The protocol uses two lossy FIFO transmission lines: the sender sends data and control information on *msgQ*, and the receiver sends acknowledgments on *ackQ*.

The complete protocol specification appears in module *AlternatingBit* in Figure 14.1 on the following two pages. It is fairly straightforward, except for the liveness condition. Because messages can be repeatedly lost from the queues, strong fairness of the actions that receive messages is required to ensure that a message that keeps getting resent is eventually received. However, don’t worry about the details of the specification. For now, all you need to know are the declarations

CONSTANT *Data* The set of data values that can be sent.

VARIABLES *msgQ*, *ackQ*, *sBit*, *sAck*, *rBit*, *sent*, *rcvd*

and the types of the variables:

MODULE *AlternatingBit*

This specification describes a protocol for using lossy FIFO transmission lines to transmit a sequence of values from a sender to a receiver. The sender sends a data value  $d$  by sending a sequence of  $\langle b, d \rangle$  messages on  $msgQ$ , where  $b$  is a control bit. It knows that the message has been received when it receives the ack  $b$  from the receiver on  $ackQ$ . It sends the next value with a different control bit. The receiver knows that a message on  $msgQ$  contains a new value when the control bit differs from the last one it has received. The receiver keeps sending the last control bit it received on  $ackQ$ .

EXTENDS *Naturals, Sequences*

CONSTANTS  $Data$  The set of data values that can be sent.

VARIABLES  $msgQ$ , The sequence of  $\langle$ control bit, data value $\rangle$  messages in transit to the receiver.

$ackQ$ , The sequence of one-bit acknowledgments in transit to the sender.

$sBit$ , The last control bit sent by sender; it is complemented when sending a new data value.

$sAck$ , The last acknowledgment bit received by the sender.

$rBit$ , The last control bit received by the receiver.

$sent$ , The last value sent by the sender.

$rcvd$ , The last value received by the receiver.

$ABInit \triangleq \begin{array}{l} \wedge msgQ = \langle \rangle \\ \wedge ackQ = \langle \rangle \\ \wedge sBit \in \{0, 1\} \\ \wedge sAck = sBit \\ \wedge rBit = sBit \\ \wedge sent \in Data \\ \wedge rcvd \in Data \end{array}$  The initial condition:  
Both message queues are empty.  
All the bits equal 0 or 1  
and are equal to each other.  
The initial values of  $sent$  and  $rcvd$   
are arbitrary data values.

$ABTypeInv \triangleq \begin{array}{l} \wedge msgQ \in Seq(\{0, 1\} \times Data) \\ \wedge ackQ \in Seq(\{0, 1\}) \\ \wedge sBit \in \{0, 1\} \\ \wedge sAck \in \{0, 1\} \\ \wedge rBit \in \{0, 1\} \\ \wedge sent \in Data \\ \wedge rcvd \in Data \end{array}$  The type-correctness invariant.

$SndNewValue(d) \triangleq \begin{array}{l} \wedge sAck = sBit \\ \wedge sent' = d \\ \wedge sBit' = 1 - sBit \\ \wedge msgQ' = Append(msgQ, \langle sBit', d \rangle) \\ \wedge \text{UNCHANGED } \langle ackQ, sAck, rBit, rcvd \rangle \end{array}$  The action in which the sender sends a new data value  $d$ .  
Enabled iff  $sAck$  equals  $sBit$ .  
Set  $sent$  to  $d$ .  
Complement control bit  $sBit$   
Send value on  $msgQ$  with new control bit.  
UNCHANGED  $\langle ackQ, sAck, rBit, rcvd \rangle$

**Figure 14.1a:** The alternating bit protocol (beginning).

|                                                                                 |                                                           |
|---------------------------------------------------------------------------------|-----------------------------------------------------------|
| $ReSndMsg \triangleq$                                                           | The sender resends the last message it sent on $msgQ$ .   |
| $\wedge sAck \neq sBit$                                                         | Enabled iff $sAck$ doesn't equal $sBit$ .                 |
| $\wedge msgQ' = Append(msgQ, \langle sBit, sent \rangle)$                       | Resend the last value in $send$ .                         |
| $\wedge \text{UNCHANGED } \langle ackQ, sBit, sAck, rBit, sent, rcvd \rangle$   |                                                           |
| $RcvMsg \triangleq$                                                             | The receiver receives the message at the head of $msgQ$ . |
| $\wedge msgQ \neq \langle \rangle$                                              | Enabled iff $msgQ$ not empty.                             |
| $\wedge msgQ' = Tail(msgQ)$                                                     | Remove message from head of $msgQ$ .                      |
| $\wedge rBit' = Head(msgQ)[1]$                                                  | Set $rBit$ to message's control bit.                      |
| $\wedge rcvd' = Head(msgQ)[2]$                                                  | Set $rcvd$ to message's data value.                       |
| $\wedge \text{UNCHANGED } \langle ackQ, sBit, sAck, sent \rangle$               |                                                           |
| $SndAck \triangleq \wedge ackQ' = Append(ackQ, rBit)$                           | The receiver sends $rBit$ on $ackQ$ at any time.          |
| $\wedge \text{UNCHANGED } \langle msgQ, sBit, sAck, rBit, sent, rcvd \rangle$   |                                                           |
| $RcvAck \triangleq \wedge ackQ \neq \langle \rangle$                            | The sender receives an ack on $ackQ$ .                    |
| $\wedge ackQ' = Tail(ackQ)$                                                     | It removes the ack and sets $sAck$ to its value.          |
| $\wedge sAck' = Head(ackQ)$                                                     |                                                           |
| $\wedge \text{UNCHANGED } \langle msgQ, sBit, rBit, sent, rcvd \rangle$         |                                                           |
| $Lose(q) \triangleq$                                                            | The action of losing a message from queue $q$ .           |
| $\wedge q \neq \langle \rangle$                                                 | Enabled iff $q$ is not empty.                             |
| $\wedge \exists i \in 1 \dots Len(q) :$                                         | For some $i$ ,                                            |
| $q' = [j \in 1 \dots (Len(q) - 1) \mapsto \text{IF } j < i \text{ THEN } q[j]$  | remove the $i^{\text{th}}$ message from $q$ .             |
| $\text{ELSE } q[j + 1]]$                                                        | Leave every variable unchanged except $msgQ$ and $ackQ$ . |
| $\wedge \text{UNCHANGED } \langle sBit, sAck, rBit, sent, rcvd \rangle$         |                                                           |
| $LoseMsg \triangleq Lose(msgQ) \wedge \text{UNCHANGED } ackQ$                   | Lose a message from $msgQ$ .                              |
| $LoseAck \triangleq Lose(ackQ) \wedge \text{UNCHANGED } msgQ$                   | Lose a message from $ackQ$ .                              |
| $ABNext \triangleq \vee \exists d \in Data : SndNewValue(d)$                    | The next-state action.                                    |
| $\vee ReSndMsg \vee RcvMsg \vee SndAck \vee RcvAck$                             |                                                           |
| $\vee LoseMsg \vee LoseAck$                                                     |                                                           |
| $abvars \triangleq \langle msgQ, ackQ, sBit, sAck, rBit, sent, rcvd \rangle$    | The tuple of all variables.                               |
| $ABFairness \triangleq \wedge WF_{abvars}(ReSndMsg) \wedge WF_{abvars}(SndAck)$ | The liveness condition.                                   |
| $\wedge SF_{abvars}(RcvMsg) \wedge SF_{abvars}(RcvAck)$                         |                                                           |
| $ABSpec \triangleq ABInit \wedge \square[ABNext]_{abvars} \wedge ABFairness$    | The complete specification.                               |
| $\text{THEOREM } ABSpec \Rightarrow \square ABTypeInv$                          |                                                           |

Figure 14.1b: The alternating bit protocol (end).

- $msgQ$  is a sequence of elements in  $\{0, 1\} \times Data$ .
- $ackQ$  is a sequence of elements in  $\{0, 1\}$ .
- $sBit$ ,  $sAck$ , and  $rBit$  are elements of  $\{0, 1\}$ .
- $sent$  and  $rcvd$  are elements of  $Data$ .

The input to TLC consists of a TLA<sup>+</sup> module and a configuration file. TLC assumes the specification has the form of formula (14.1) on page 221. The configuration file tells TLC the names of the specification and of the properties to be checked. For example, the configuration file for the alternating bit protocol will contain the declaration

**SPECIFICATION ABSpec**

telling TLC to take *ABSpec* as the specification. If your specification has the form  $Init \wedge \square[Next]_{vars}$ , with no liveness condition, then instead of using a **SPECIFICATION** statement, you can declare the initial predicate and next-state action by putting the following two statements in the configuration file:

**INIT Init**  
**NEXT Next**

The property or properties to be checked are specified with a **PROPERTY** statement. For example, to check that *ABTypeInv* is actually an invariant, we could have TLC check that the specification implies  $\square ABTypeInv$  by adding the definition

$InvProperty \triangleq \square ABTypeInv$

to module *AlternatingBit* and putting the statement

**PROPERTY InvProperty**

in the configuration file. Invariance checking is so common that TLC allows you instead to put the following statement in the configuration file:

**INVARIANT ABTypeInv**

The **INVARIANT** statement must specify a state predicate. To check invariance with a **PROPERTY** statement, the specified property has to be of the form  $\square P$ . Specifying a state predicate  $P$  in a **PROPERTY** statement tells TLC to check that the specification implies  $P$ , meaning that  $P$  is true in the initial state of every behavior satisfying the specification.

TLC works by generating behaviors that satisfy the specification. To do this, it must be given what we call a *model* of the specification. To define a model, we must assign values to the specification's constant parameters. The only constant parameter of the alternating bit protocol specification is the set

*Data* of data values. We can tell TLC to let *Data* equal the set containing two arbitrary elements, named *d1* and *d2*, by putting the following declaration in the configuration file:

```
CONSTANT Data = {d1, d2}
```

(We can use any sequence of letters and digits containing at least one letter as the name of an element.)

There are two ways to use TLC. The default method is *model checking*, in which it tries to find all reachable states—that is, all states<sup>1</sup> that can occur in behaviors satisfying the formula  $Init \wedge \square[Next]_{vars}$ . You can also run TLC in *simulation* mode, in which it randomly generates behaviors, without trying to check all reachable states. We now consider model checking; simulation mode is described in Section 14.3.2 on page 243.

Exhaustively checking all reachable states is impossible for the alternating bit protocol because the sequences of messages can get arbitrarily long, so there are infinitely many reachable states. We must further constrain the model to make it finite—that is, so it allows only a finite number of possible states. We do this by defining a state predicate called the *constraint* that asserts bounds on the lengths of the sequences. For example, the following constraint asserts that *msgQ* and *ackQ* have length at most 2:

$$\begin{aligned} & \wedge \text{Len}(\text{msgQ}) \leq 2 \\ & \wedge \text{Len}(\text{ackQ}) \leq 2 \end{aligned}$$

Instead of specifying the bounds on the lengths of sequences in this way, I prefer to make them parameters and to assign them values in the configuration file. We don’t want to put into the specification itself declarations and definitions that are just for TLC’s benefit. So, we write a new module, called *MCAlternatingBit*, that extends the *AlternatingBit* module and can be used as input to TLC. This module appears in Figure 14.2 on the next page. A possible configuration file for the module appears in Figure 14.3 on the next page. Observe that the configuration file must specify values for all the constant parameters of the specification—in this case, the parameter *Data* from the *AlternatingBit* module and the two parameters declared in module *MCAlternatingBit* itself. You can put comments in the configuration file, using the TLA<sup>+</sup> comment syntax described in Section 3.5 (page 32).

When a constraint *Constr* is specified, TLC checks every state that appears in a behavior satisfying  $Init \wedge \square[Next]_{vars} \wedge \square\text{Constr}$ . In the rest of this chapter, these states will be called the *reachable* ones.

The keywords  
CONSTANT and  
CONSTANTS are  
equivalent, as are  
INVARIANT and  
INVARIANTS.

Section 14.3 below describes how actions as well as state predicates can be used as constraints.

<sup>1</sup>As explained in Section 2.3 (page 18), a state is an assignment of values to all possible variables. However, when discussing a particular specification, we usually consider a state to be an assignment of values to that specification’s variables. That’s what I’m doing in this chapter.

---

MODULE *MCA*lternatingBit

---

EXTENDS *AlternatingBit*

CONSTANTS *msgQLen*, *ackQLen*

*SeqConstraint*  $\triangleq$   $\wedge \text{Len}(\text{msgQ}) \leq \text{msgQLen}$   
 $\wedge \text{Len}(\text{ackQ}) \leq \text{ackQLen}$

A constraint on the lengths of sequences for use by TLC.

**Figure 14.2:** Module *MCAlternatingBit*.

Having TLC check the type invariant will catch many simple mistakes. When we've corrected all the errors we can find that way, we then want to look for less obvious ones. A common error is for an action not to be enabled when it should be, preventing some states from being reached. You can discover if an action is never enabled by using the *coverage* option, described on page 252. To discover if an action is just sometimes incorrectly disabled, try checking liveness properties. An obvious liveness property for the alternating bit protocol is that every message sent is eventually delivered. A message  $d$  has been sent when  $sent = d$  and  $sBit \neq sAck$ . So, a naive way to state this property is

The temporal operator  $\leadsto$  is defined on page 91.

$$SentLeadsToRcvd \triangleq \forall d \in Data : (sent = d) \wedge (sBit \neq sAck) \rightsquigarrow (rcvd = d)$$

Formula *SentLeadsToRcvd* asserts that, for any data value  $d$ , if *sent* ever equals  $d$  when *sBit* does not equal *sAck*, then *rcvd* must eventually equal  $d$ . This doesn't assert that every message sent is eventually delivered. For example, it is satisfied by a behavior in which a particular value  $d$  is sent twice, but received only once. However, the formula is good enough for our purposes because the protocol doesn't depend on the actual values being sent. If it were possible for the same value to be sent twice but received only once, then it would be possible for two different values to be sent and only one received, violating *SentLeadsToRcvd*. We therefore add the definition of *SentLeadsToRcvd* to module *MCAlternatingBit* and add the following statement to the configuration file:

PROPERTY SentLeadsToRcvd

```
CONSTANTS Data = {d1, d2} (* Is this big enough? *)
 msgQLen = 2
 ackQLen = 2 * Try 3 next.
SPECIFICATION ABSpec
INVARIANT ABTypeInv
CONSTRAINT SeqConstraint
```

**Figure 14.3:** A configuration file for module *MCAlternatingBit*.

Checking liveness properties is a lot slower than other kinds of checking, so you should do it only after you've found all the errors you can by checking invariance properties.

Checking type correctness and property *SentLeadsToRcvd* is a good way to start looking for errors. But ultimately, we would like to see if the protocol meets its specification. However, we don't have its specification. In fact, it is typical in practice that we are called upon to check the correctness of a system design without any formal specification of what the system is supposed to do. In that case, we can write an *ex post facto* specification. Module *ABCorrectness* in Figure 14.4 on the next page is such a specification of correctness for the alternating bit protocol. It is actually a simplified version of the protocol's specification in which, instead of being read from messages, the variables *rcvd*, *rBit*, and *sAck* are obtained directly from the variables of the other process.

We want to check that the specification *ABSpec* of module *AlternatingBit* implies formula *ABCSpec* of module *ABCorrectness*. To do this, we modify module *MCAlternatingBit* by adding the statement

INSTANCE *ABCorrectness*

and we modify the PROPERTY statement of the configuration file to

PROPERTIES *ABCSpec* *SentLeadsToRcvd*

This example is atypical because the correctness specification *ABCSpec* does not involve variable hiding (temporal existential quantification). Let's now suppose module *ABCorrectness* did declare another variable *h* that appeared in *ABCSpec*, and that the correctness condition for the alternating bit protocol was *ABCSpec* with *h* hidden. The correctness condition would then be expressed formally in TLA<sup>+</sup> as follows:

$$\begin{aligned} AB(h) &\triangleq \text{INSTANCE } \textit{ABCorrectness} \\ \text{THEOREM } \textit{ABSpec} &\Rightarrow \exists h : AB(h)! \textit{ABCSpec} \end{aligned}$$

TLC could not check this theorem directly because it cannot handle the temporal existential quantifier  $\exists$ . We would check this theorem with TLC the same way we would try to prove it—namely, by using a refinement mapping. As explained in Section 5.8 on page 62, we would define a state function *oh* in terms of the variables of module *AlternatingBit* and we would prove

$$(14.2) \textit{ABSpec} \Rightarrow AB(oh)! \textit{ABCSpec}$$

To get TLC to check this theorem, we would add the definition

$$\textit{ABCSpecBar} \triangleq AB(oh)! \textit{ABCSpec}$$

and have TLC check the property *ABCSpecBar*.

The keywords PROPERTY and PROPERTIES are equivalent.

This use of INSTANCE is explained in Section 4.3 (page 41).

---

MODULE *ABCorrectness*

---

```

EXTENDS Naturals
CONSTANTS Data
VARIABLES sBit, sAck, rBit, sent, rcvd

ABCInit \triangleq \wedge sBit $\in \{0, 1\}
 \wedge sAck = sBit
 \wedge rBit = sBit
 \wedge sent \in Data
 \wedge rcvd \in Data

CSndNewValue(d) \triangleq \wedge sAck = sBit
 \wedge sent' = d
 \wedge sBit' = $1 - \text{sBit}$
 \wedge UNCHANGED $\langle \text{sAck}, \text{rBit}, \text{rcvd} \rangle$

CRcvMsg \triangleq \wedge rBit \neq sBit
 \wedge rBit' = sBit
 \wedge rcvd' = sent
 \wedge UNCHANGED $\langle \text{sBit}, \text{sAck}, \text{sent} \rangle$

CRcvAck \triangleq \wedge rBit \neq sAck
 \wedge sAck' = rBit
 \wedge UNCHANGED $\langle \text{sBit}, \text{rBit}, \text{sent}, \text{rcvd} \rangle$

ABCNext \triangleq $\vee \exists d \in \text{Data} : \text{CSndNewValue}(d)$
 $\vee \text{CRcvMsg} \vee \text{CRcvAck}$

cvars \triangleq $\langle \text{sBit}, \text{sAck}, \text{rBit}, \text{sent}, \text{rcvd} \rangle$

ABCFairness \triangleq WFcvars(CRcvMsg) \wedge WFcvars(CRcvAck)

ABCSpec \triangleq ABCInit \wedge $\square[\text{ABCNext}]_{\text{cvars}} \wedge \text{ABCFairness}$$
```

---

**Figure 14.4:** A specification of correctness of the alternating bit protocol.

When TLC checks a property, it does not actually verify that the specification implies the property. Instead, it checks that (i) the safety part of the specification implies the safety part of the property and (ii) the specification implies the liveness part of the property. For example, suppose that the specification *Spec* and the property *Prop* are

$$\begin{aligned}
\text{Spec} &\triangleq \text{Init} \wedge \square[\text{Next}]_{\text{vars}} \wedge \text{Temporal} \\
\text{Prop} &\triangleq \text{ImpliedInit} \wedge \square[\text{ImpliedAction}]_{\text{pvars}} \wedge \text{ImpliedTemporal}
\end{aligned}$$

where *Temporal* and *ImpliedTemporal* are liveness properties. In this case, TLC checks the two formulas

$$\begin{aligned} \text{Init} \wedge \square[\text{Next}]_{\text{vars}} &\Rightarrow \text{ImpliedInit} \wedge \square[\text{ImpliedAction}]_{\text{pvars}} \\ \text{Spec} &\Rightarrow \text{ImpliedTemporal} \end{aligned}$$

This means that you cannot use TLC to check that a non-machine-closed specification satisfies a safety property. (Machine closure is discussed in Section 8.9.2 on page 111.) Section 14.3 below more precisely describes how TLC checks properties.

## 14.2 What TLC Can Cope With

No model checker can handle all the specifications that we can write in a language as expressive as TLA<sup>+</sup>. However, TLC seems able to handle most TLA<sup>+</sup> specifications that people actually write. Getting TLC to handle a specification may require a bit of trickery, but it can usually be done without having to make any changes to the specification itself.

This section explains what TLC can and cannot cope with, and gives some ways to make it cope. The best way to understand TLC’s limitations is to understand how it works. So, this section describes how TLC “executes” a specification.

### 14.2.1 TLC Values

A state is an assignment of values to variables. TLA<sup>+</sup> allows you to describe a wide variety of values—for example, the set of all sequences of prime numbers. TLC can compute only a restricted class of values, called TLC values. Those values are built from the following four types of primitive values:

*Booleans* The values TRUE and FALSE.

*Integers* Values like 3 and -1.

*Strings* Values like “ab3”.

*Model Values* These are values introduced in the CONSTANT statement of the configuration file. For example, the configuration file shown in Figure 14.3 on page 227 introduces the model values d1 and d2. Model values with different names are assumed to be different.

A TLC value is defined inductively to be either

1. a primitive value, or

2. a finite set of comparable TLC values (*comparable* is defined below), or
3. a function  $f$  whose domain is a TLC value such that  $f[x]$  is a TLC value, for all  $x$  in DOMAIN  $f$ .

For example, the first two rules imply that

$$(14.3) \{ \{ "a", "b" \}, \{ "b", "c" \}, \{ "c", "d" \} \}$$

is a TLC value because rules 1 and 2 imply that  $\{ "a", "b" \}$ ,  $\{ "b", "c" \}$ , and  $\{ "c", "d" \}$  are TLC values, and the second rule then implies that (14.3) is a TLC value. Since tuples and records are functions, rule 3 implies that a record or tuple whose components are TLC values is a TLC value. For example,  $\langle 1, "a", 2, "b" \rangle$  is a TLC value.

To complete the definition of what a TLC value is, I must explain what *comparable* means in rule 2. The basic idea is that two values should be comparable iff the semantics of TLA<sup>+</sup> determines whether or not they are equal. For example, strings and numbers are not comparable because the semantics of TLA<sup>+</sup> doesn't tell us whether or not "abc" equals 42. The set  $\{ "abc", 42 \}$  is therefore not a TLC value; rule 2 doesn't apply because "abc" and 42 are not comparable. On the other hand,  $\{ "abc" \}$  and  $\{ 4, 2 \}$  are comparable because sets having different numbers of elements must be unequal. Hence, the two-element set  $\{ \{ "abc" \}, \{ 4, 2 \} \}$  is a TLC value. TLC considers a model value to be comparable to, and unequal to, any other value. The precise rules for comparability are given in Section 14.7.2.

### 14.2.2 How TLC Evaluates Expressions

Checking a specification requires evaluating expressions. For example, TLC does invariance checking by evaluating the invariant in each reachable state—that is, computing its TLC value, which should be TRUE. To understand what TLC can and cannot do, you have to know how it evaluates expressions.

TLC evaluates expressions in a straightforward way, generally evaluating subexpressions "from left to right". In particular:

- It evaluates  $p \wedge q$  by first evaluating  $p$  and, if it equals TRUE, then evaluating  $q$ .
- It evaluates  $p \vee q$  by first evaluating  $p$  and, if it equals FALSE, then evaluating  $q$ . It evaluates  $p \Rightarrow q$  as  $\neg p \vee q$ .
- It evaluates IF  $p$  THEN  $e_1$  ELSE  $e_2$  by first evaluating  $p$ , then evaluating either  $e_1$  or  $e_2$ .

To understand the significance of these rules, let's consider a simple example. TLC cannot evaluate the expression  $x[1]$  if  $x$  equals  $\langle \rangle$ , since  $\langle \rangle[1]$  is silly. (The

empty sequence  $\langle \rangle$  is a function whose domain is the empty set and hence does not contain 1.) The first rule implies that, if  $x$  equals  $\langle \rangle$ , then TLC can evaluate the formula

$$(x \neq \langle \rangle) \wedge (x[1] = 0)$$

but not the (logically equivalent) formula

$$(x[1] = 0) \wedge (x \neq \langle \rangle)$$

(When evaluating the latter formula, TLC first tries to compute  $\langle \rangle[1] = 0$ , reporting an error because it can't.) Fortunately, we naturally write the first formula rather than the second because it's easier to understand. People understand a formula by “mentally evaluating” it from left to right, much the way TLC does.

TLC evaluates  $\exists x \in S : p$  by enumerating the elements  $s_1, \dots, s_n$  of  $S$  in some order and then evaluating  $p$  with  $s_i$  substituted for  $x$ , successively for  $i = 1, \dots, n$ . It enumerates the elements of a set  $S$  in a very straightforward way, and it gives up and declares an error if the set is not obviously finite. For example, it can obviously enumerate the elements of  $\{0, 1, 2, 3\}$  and  $0 \dots 3$ . It enumerates a set of the form  $\{x \in S : p\}$  by first enumerating  $S$ , so it can enumerate  $\{i \in 0 \dots 5 : i < 4\}$  but not  $\{i \in \text{Nat} : i < 4\}$ .

TLC evaluates the expressions  $\forall x \in S : p$  and  $\text{CHOOSE } x \in S : p$  by first enumerating the elements of  $S$ , much the same way as it evaluates  $\exists x \in S : p$ . The semantics of  $\text{TLA}^+$  states that  $\text{CHOOSE } x \in S : p$  is an arbitrary value if there is no  $x$  in  $S$  for which  $p$  is true. However, this case almost always arises because of a mistake, so TLC treats it as an error. Note that evaluating the expression

IF  $n > 5$  THEN  $\text{CHOOSE } i \in 1 \dots n : i > 5$  ELSE 42

will not produce an error because TLC will not evaluate the  $\text{CHOOSE}$  expression if  $n \leq 5$ . (TLC would report an error if it tried to evaluate the  $\text{CHOOSE}$  expression when  $n \leq 5$ .)

TLC cannot evaluate “unbounded” quantifiers or  $\text{CHOOSE}$  expressions—that is, expressions having one of the forms

$$\exists x : p \quad \forall x : p \quad \text{CHOOSE } x : p$$

TLC cannot evaluate any expression whose value is not a TLC value, as defined in Section 14.2.1 above. In particular, TLC can evaluate a set-valued expression only if that expression equals a finite set, and it can evaluate a function-valued expression only if that expression equals a function whose domain is a finite set. TLC will evaluate expressions of the following forms only if it can enumerate the set  $S$ :

$$\begin{array}{lll} \exists x \in S : p & \forall x \in S : p & \text{CHOOSE } x \in S : p \\ \{x \in S : p\} & \{e : x \in S\} & [x \in S \mapsto e] \\ \text{SUBSET } S & \text{UNION } S & \end{array}$$

TLC can often evaluate an expression even when it can't evaluate all subexpressions. For example, it can evaluate

$$[n \in Nat \mapsto n * (n + 1)][3]$$

which equals the TLC value 12, even though it can't evaluate

$$[n \in Nat \mapsto n * (n + 1)]$$

which equals a function whose domain is the set *Nat*. (A function can be a TLC value only if its domain is a finite set.)

TLC evaluates recursively defined functions with a simple recursive procedure. If  $f$  is defined by  $f[x \in S] \triangleq e$ , then TLC evaluates  $f[c]$  by evaluating  $e$  with  $c$  substituted for  $x$ . This means that it can't handle some legal function definitions. For example, consider this definition from page 68:

$$\begin{aligned} mr[n \in Nat] &\triangleq \\ &[f \mapsto \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } mr[n - 1].f * mr[n].g, \\ &\quad g \mapsto \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } mr[n - 1].f + mr[n - 1].g] \end{aligned}$$

To evaluate  $mr[3]$ , TLC substitutes 3 for  $n$  and starts evaluating the right-hand side. But because  $mr[n]$  appears in the right-hand side, TLC must evaluate the subexpression  $mr[3]$ , which it does by substituting 3 for  $n$  and starting to evaluate the right-hand side. And so on. TLC eventually detects that it's in an infinite loop and reports an error.

Legal recursive definitions that cause TLC to loop like this are rare, and they can be rewritten so TLC can handle them. Recall that we defined  $mr$  to express the mutual recursion:

$$\begin{aligned} f[n] &= \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } f[n - 1] * g[n] \\ g[n] &= \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } f[n - 1] + g[n - 1] \end{aligned}$$

The subexpression  $mr[n]$  appeared in the expression defining  $mr[n]$  because  $f[n]$  depends on  $g[n]$ . To eliminate it, we have to rewrite the mutual recursion so that  $f[n]$  depends only on  $f[n - 1]$  and  $g[n - 1]$ . We do this by expanding the definition of  $g[n]$  in the expression for  $f[n]$ . Since the ELSE clause applies only to the case  $n \neq 0$ , we can rewrite the expression for  $f[n]$  as

$$f[n] = \text{IF } n = 0 \text{ THEN } 17 \text{ ELSE } f[n - 1] * (f[n - 1] + g[n - 1])$$

This leads to the following equivalent definition of  $mr$ :

$$\begin{aligned} mr[n \in Nat] &\triangleq \\ &[f \mapsto \text{IF } n = 0 \text{ THEN } 17 \\ &\quad \text{ELSE } mr[n - 1].f * (mr[n - 1].f + mr[n - 1].g), \\ &\quad g \mapsto \text{IF } n = 0 \text{ THEN } 42 \text{ ELSE } mr[n - 1].f + mr[n - 1].g] \end{aligned}$$

With this definition, TLC has no trouble evaluating  $mr[3]$ .

The evaluation of ENABLED predicates and the action-composition operator “ $\cdot$ ” are described on page 240 in Section 14.2.6. Section 14.3 explains how TLC evaluates temporal-logic formulas for temporal checking.

If you’re not sure whether TLC can evaluate an expression, try it and see. But don’t wait until TLC gets to the expression in the middle of checking the entire specification. Instead, make a small example in which TLC evaluates just that expression. See the explanation on page 14.5.3 of how to use TLC as a TLA<sup>+</sup> calculator.

### 14.2.3 Assignment and Replacement

As we saw in the alternating bit example, the configuration file must determine the value of each constant parameter. To assign a TLC value  $v$  to a constant parameter  $c$  of the specification, we write  $c = v$  in the configuration file’s CONSTANT statement. The value  $v$  may be a primitive TLC value or a finite set of primitive TLC values written in the form  $\{v_1, \dots, v_n\}$ —for example,  $\{1, -3, 2\}$ . In  $v$ , any sequence of characters like `a1` or `foo` that is not a number, a quoted string, or TRUE or FALSE is taken to be a model value.

In the assignment  $c = v$ , the symbol  $c$  need not be a constant parameter; it can also be a defined symbol. This assignment causes TLC to ignore the actual definition of  $c$  and to take  $v$  to be its value. Such an assignment is often used when TLC cannot compute the value of  $c$  from its definition. In particular, TLC cannot compute the value of  $NotAnS$  from the definition

$$NotAnS \triangleq \text{CHOOSE } n : n \notin S$$

because it cannot evaluate the unbounded CHOOSE expression. You can override this definition by assigning  $NotAnS$  a value in the CONSTANT statement of the configuration file. For example, the assignment

$$\text{NotAnS} = \text{NS}$$

causes TLC to assign to  $NotAnS$  the model value `NS`. TLC ignores the actual definition of  $NotAnS$ . If you used the name  $NotAnS$  in the specification, you’d probably want TLC’s error messages to call it `NotAnS` rather than `NS`. So, you’d probably use the assignment

$$\text{NotAnS} = \text{NotAnS}$$

which assigns to the symbol  $NotAnS$  the model value `NotAnS`. Remember that, in the assignment  $c = v$ , the symbol  $c$  must be defined or declared in the TLA<sup>+</sup> module, and  $v$  must be a primitive TLC value or a finite set of such values.

The CONSTANT statement of the configuration file can also contain *replacements* of the form  $c \leftarrow d$ , where  $c$  and  $d$  are symbols defined in the TLA<sup>+</sup>

Note that  $d$  is a defined symbol in the replacement  $c \leftarrow d$ , while  $v$  is a TLC value in the substitution  $c = v$ .

module. This causes TLC to replace  $c$  by  $d$  when performing its calculations. One use of replacement is to give a value to an operator parameter. For example, suppose we wanted to use TLC to check the write-through cache specification of Section 5.6 (page 54). The *WriteThroughCache* module extends the *MemoryInterface* module, which contains the declaration

CONSTANTS  $Send(\_, \_, \_, \_, \_), Reply(\_, \_, \_, \_, \_), \dots$

We have to tell TLC how to evaluate the operators *Send* and *Reply*. We do this by first writing a module *MCWriteThroughCache* that extends the *WriteThroughCache* module and defines two operators

$$\begin{aligned} MCSend(p, d, old, new) &\triangleq \dots \\ MCReply(p, d, old, new) &\triangleq \dots \end{aligned}$$

We then add to the configuration file's CONSTANT statement the replacements

```
Send <- MCSend
Reply <- MCReply
```

A replacement can also replace one defined symbol by another. In a specification, we usually write the simplest possible definitions. A simple definition is not always the easiest one for TLC to use. For example, suppose our specification requires an operator *Sort* such that  $Sort(S)$  is a sequence containing the elements of  $S$  in increasing order, if  $S$  is a finite set of numbers. Our specification in module *SpecMod* might use the simple definition

$$\begin{aligned} Sort(S) &\triangleq \text{CHOOSE } s \in [1 \dots \text{Cardinality}(S) \rightarrow S] : \\ &\quad \forall i, j \in \text{DOMAIN } s : (i < j) \Rightarrow (s[i] < s[j]) \end{aligned}$$

To evaluate  $Sort(S)$  for a set  $S$  containing  $n$  elements, TLC has to enumerate the  $n^n$  elements in the set  $[1 \dots n \rightarrow S]$  of functions. This may be unacceptably slow. We can write a module *MCSpecMod* that extends *SpecMod* and defines *FastSort* so it equals *Sort* when applied to finite sets of numbers, but can be evaluated more efficiently by TLC. We can then run TLC with a configuration file containing the replacement

```
Sort <- FastSort
```

One possible definition of *FastSort* is given in Section 14.4, on page 250.

#### 14.2.4 Evaluating Temporal Formulas

Section 14.2.2 (page 231) explains what kind of ordinary expressions TLC can evaluate. The specification and properties that TLC checks are temporal formulas; this section describes the class of temporal formulas it can handle.

TLC can evaluate a TLA temporal formula iff (i) the formula is *nice*—a term defined in the next paragraph—and (ii) TLC can evaluate all the ordinary expressions of which the formula is composed. For example, a formula of the form  $P \rightsquigarrow Q$  is nice, so TLC can evaluate it iff it can evaluate  $P$  and  $Q$ . (Section 14.3 below explains on what states and pairs of states TLC evaluates the component expressions of a temporal formula.)

A temporal formula is nice iff it is the conjunction of formulas that belong to one of the following four classes:

### State Predicate

**Invariance Formula** A formula of the form  $\Box P$ , where  $P$  is a state predicate.

**Box-Action Formula** A formula of the form  $\Box[A]_v$ , where  $A$  is an action and  $v$  is a state function.

**Simple Temporal Formula** To define this class, we first make the following definitions:

- The *simple Boolean operators* consist of the operators

$$\wedge \quad \vee \quad \neg \quad \Rightarrow \quad \equiv \quad \text{TRUE} \quad \text{FALSE}$$

of propositional logic together with quantification over finite, constant sets.

- A *temporal state formula* is one obtained from state predicates by applying simple Boolean operators and the temporal operators  $\Box$ ,  $\Diamond$ , and  $\rightsquigarrow$ . For example, if  $N$  is a constant, then

$$\forall i \in 1 \dots N : \Box((x = i) \Rightarrow \exists j \in 1 \dots i : \Diamond(y = j))$$

is a temporal state formula.

- A *simple action formula* is one of the following, where  $A$  is an action and  $v$  a state function:

$$\text{WF}_v(A) \quad \text{SF}_v(A) \quad \Box\Diamond\langle A \rangle_v \quad \Diamond\Box[A]_v$$

The component expressions of  $\text{WF}_v(A)$  and  $\text{SF}_v(A)$  are  $\langle A \rangle_v$  and  $\text{ENABLED} \langle A \rangle_v$ . (The evaluation of  $\text{ENABLED}$  formulas is described on page 240.)

A simple temporal formula is then defined to be one constructed from temporal state formulas and simple action formulas by applying simple Boolean operators.

For convenience, we exclude invariance formulas from the class of simple temporal formulas, so these four classes of nice temporal formulas are disjoint.

TLC can therefore evaluate the temporal formula

$$\forall i \in 1 \dots N : \Diamond(y = i) \Rightarrow \text{WF}_y((y' = y + 1) \wedge (y \geq i))$$

The terminology used here is not standard.

if  $N$  is a constant, because this is a simple temporal formula (and hence nice) and TLC can evaluate all of its component expressions. TLC cannot evaluate  $\diamond \langle x' = 1 \rangle_x$ , since this is not a nice formula. It cannot evaluate the formula  $\text{WF}_x(x'[1] = 0)$  if it must evaluate the action  $\langle x'[1] = 0 \rangle_x$  on a step  $s \rightarrow t$  in which  $x = \langle \rangle$  in state  $t$ .

A PROPERTY statement can specify any formulas that TLC can evaluate. The formula of a SPECIFICATION statement must contain exactly one conjunct that is a box-action formula. That conjunct specifies the next-state action.

### 14.2.5 Overriding Modules

TLC cannot compute  $2 + 2$  from the definition of  $+$  contained in the standard *Naturals* module. Even if we did use a definition of  $+$  from which TLC could compute sums, it would not do so very quickly. Arithmetic operators like  $+$  are implemented directly in Java, the language in which TLC is written. This is achieved by a general mechanism of TLC that allows a module to be overridden by a Java class that implements the operators defined in the module. When TLC encounters an EXTENDS *Naturals* statement, it loads the Java class that overrides the *Naturals* module rather than reading the module itself. There are Java classes to override the following standard modules: *Naturals*, *Integers*, *Sequences*, *FiniteSets*, and *Bags*. (The *TLC* module described below in Section 14.4 is also overridden by a Java class.) Intrepid Java programmers will find that writing a Java class to override a module is not too hard.

### 14.2.6 How TLC Computes States

When TLC evaluates an invariant, it is calculating the invariant's value, which is either TRUE or FALSE. When TLC evaluates the initial predicate or the next-state action, it is computing a set of states—for the initial predicate, the set of all initial states, and for the next-state action, the set of possible *successor states* (primed states) reached from a given starting (unprimed) state. I will describe how TLC does this for the next-state action; the evaluation of the initial predicate is analogous.

Recall that a state is an assignment of values to variables. TLC computes the successors of a given state  $s$  by assigning to all unprimed variables their values in state  $s$ , assigning no values to the primed variables, and then evaluating the next-state action. TLC evaluates the next-state action as described in Section 14.2.2 (page 231), except for two differences, which I now describe. This description assumes that TLC has already performed all the assignments and replacements specified by the CONSTANT statement of the configuration file and has expanded all definitions. Thus, the next-state action is a formula containing only variables, primed variables, model values, and built-in TLA<sup>+</sup> operators and constants.

The first difference in evaluating the next-state action is that TLC does not evaluate disjunctions from left to right. Instead, when it evaluates a subformula  $A_1 \vee \dots \vee A_n$ , it splits the computation into  $n$  separate evaluations, each taking the subformula to be one of the  $A_i$ . Similarly, when it evaluates  $\exists x \in S : p$ , it splits the computation into separate evaluations for each element of  $S$ . An implication  $P \Rightarrow Q$  is treated as the disjunction  $(\neg P) \vee Q$ . For example, TLC splits the evaluation of

$$(A \Rightarrow B) \vee (C \wedge (\exists i \in S : D(i)) \wedge E)$$

into separate evaluations of the three disjuncts  $\neg A$ ,  $B$ , and

$$C \wedge (\exists i \in S : D(i)) \wedge E$$

To evaluate the latter disjunct, it first evaluates  $C$ . If it obtains the value TRUE, then it splits this evaluation into the separate evaluations of  $D(i) \wedge E$ , for each  $i$  in  $S$ . It evaluates  $D(i) \wedge E$  by first evaluating  $D(i)$  and, if it obtains the value TRUE, then evaluating  $E$ .

The second difference in the way TLC evaluates the next-state action is that, for any variable  $x$ , if it evaluates an expression of the form  $x' = e$  when  $x'$  has not yet been assigned a value, then the evaluation yields the value TRUE and TLC assigns to  $x'$  the value obtained by evaluating the expression  $e$ . TLC evaluates an expression of the form  $x' \in S$  as if it were  $\exists v \in S : x' = v$ . It evaluates UNCHANGED  $x$  as  $x' = x$  for any variable  $x$ , and UNCHANGED  $\langle e_1, \dots, e_n \rangle$  as

$$(\text{UNCHANGED } e_1) \wedge \dots \wedge (\text{UNCHANGED } e_n)$$

for any expressions  $e_i$ . Hence, TLC evaluates UNCHANGED  $\langle x, \langle y, z \rangle \rangle$  as if it were

$$(x' = x) \wedge (y' = y) \wedge (z' = z)$$

Except when evaluating an expression of the form  $x' = e$ , TLC reports an error if it encounters a primed variable that has not yet been assigned a value. An evaluation stops, finding no states, if a conjunct evaluates to FALSE. An evaluation that completes and obtains the value TRUE finds the state determined by the values assigned to the primed variables. In the latter case, TLC reports an error if some primed variable has not been assigned a value.

To illustrate how this works, let's consider how TLC evaluates the next-state action

$$\begin{aligned} (14.4) \vee \wedge x' &\in 1 \dots \text{Len}(y) \\ &\wedge y' = \text{Append}(\text{Tail}(y), x') \\ &\vee \wedge x' = x + 1 \\ &\wedge y' = \text{Append}(y, x') \end{aligned}$$

We first consider the starting state with  $x = 1$  and  $y = \langle 2, 3 \rangle$ . TLC splits the computation into evaluating the two disjuncts separately. It begins evaluating

the first disjunct of (14.4) by evaluating its first conjunct, which it treats as  $\exists i \in 1 \dots \text{Len}(y) : x' = i$ . Since  $\text{Len}(y) = 2$ , the evaluation splits into separate evaluations of

$$(14.5) \quad \begin{array}{l} \wedge x' = 1 \\ \wedge y' = \text{Append}(\text{Tail}(y), x') \end{array} \quad \begin{array}{l} \wedge x' = 2 \\ \wedge y' = \text{Append}(\text{Tail}(y), x') \end{array}$$

TLC evaluates the first of these actions as follows. It evaluates the first conjunct, obtaining the value TRUE and assigning to  $x'$  the value 1; it then evaluates the second conjunct, obtaining the value TRUE and assigning to  $y'$  the value  $\text{Append}(\text{Tail}(\langle 2, 3 \rangle), 1)$ . So, evaluating the first action of (14.5) finds the successor state with  $x = 1$  and  $y = \langle 3, 1 \rangle$ . Similarly, evaluating the second action of (14.5) finds the successor state with  $x = 2$  and  $y = \langle 3, 2 \rangle$ . In a similar way, TLC evaluates the second disjunct of (14.4) to find the successor state with  $x = 2$  and  $y = \langle 2, 3, 2 \rangle$ . Hence, the evaluation of (14.4) finds three successor states.

Next, consider how TLC evaluates the next-state action (14.4) in a state with  $x = 1$  and  $y$  equal to the empty sequence  $\langle \rangle$ . Since  $\text{Len}(y) = 0$  and  $1 \dots 0$  is the empty set  $\{ \}$ , TLC evaluates the first disjunct as

$$\begin{array}{l} \wedge \exists i \in \{ \} : x' = i \\ \wedge y' = \text{Append}(\text{Tail}(y), x') \end{array}$$

Evaluating the first conjunct yields FALSE, so the evaluation of the first disjunct of (14.4) stops, finding no successor states. Evaluating the second disjunct yields the successor state with  $x = 2$  and  $y = \langle 2 \rangle$ .

Since TLC evaluates conjuncts from left to right, their order can affect whether or not TLC can evaluate the next-state action. For example, suppose the two conjuncts in the first disjunct of (14.4) were reversed, like this:

$$\begin{array}{l} \wedge y' = \text{Append}(\text{Tail}(y), x') \\ \wedge x' \in 1 \dots \text{Len}(y) \end{array}$$

When TLC evaluates the first conjunct of this action, it encounters the expression  $\text{Append}(\text{Tail}(y), x')$  before it has assigned a value to  $x'$ , so it reports an error. Moreover, even if we were to change that  $x'$  to an  $x$ , TLC could still not evaluate the action starting in a state with  $y = \langle \rangle$ , since it would encounter the silly expression  $\text{Tail}(\langle \rangle)$  when evaluating the first conjunct.

The description given above of how TLC evaluates an arbitrary next-state action is good enough to explain how it works in almost all cases that arise in practice. However, it is not completely accurate. For example, interpreted literally, it would imply that TLC can cope with the following two next-state actions, which are both logically equivalent to  $(x' = \text{TRUE}) \wedge (y' = 1)$ :

$$(14.6) \quad (x' = (y' = 1)) \wedge (x' = \text{TRUE}) \quad \text{IF } x' = \text{TRUE} \text{ THEN } y' = 1 \text{ ELSE FALSE}$$

In fact, TLC will produce error messages when presented with either of these bizarre next-state actions.

Remember that TLC computes initial states by using a similar procedure to evaluate the initial predicate. Instead of starting from given values of the unprimed variables and assigning values to the primed variables, it assigns values to the unprimed variables.

TLC evaluates ENABLED formulas essentially the same way it evaluates a next-state action. More precisely, to evaluate a formula  $\text{ENABLED } A$ , TLC computes successor states as if  $A$  were the next-state action. The formula evaluates to TRUE iff there exists a successor state. To check if a step  $s \rightarrow t$  satisfies the composition  $A \cdot B$  of actions  $A$  and  $B$ , TLC first computes all states  $u$  such that  $s \rightarrow u$  is an  $A$  step and then checks if  $u \rightarrow t$  is a  $B$  step for some such  $u$ .

TLC may also have to evaluate an action when checking a property. In that case, it evaluates the action as it would any expression, and it has no trouble evaluating even the bizarre actions (14.6).

Action composition is discussed on page 77.

## 14.3 How TLC Checks Properties

Section 14.2 above explains how TLC evaluates expressions and computes initial states and successor states. This section describes how TLC uses evaluation to check properties—first for model-checking mode (its default), and then for simulation mode.

First, let's define some formulas that are obtained from the configuration file. In these definitions, a *specification conjunct* is a conjunct of the formula named by the **SPECIFICATION** statement (if there is one), a *property conjunct* is a conjunct of a formula named by a **PROPERTY** statement, and the conjunction of an empty set of formulas is defined to be TRUE. The definitions use the four classes of nice temporal formulas defined above in Section 14.2.4 on page 235.

*Init* The specification's initial state predicate. It is specified by an **INIT** or **SPECIFICATION** statement. In the latter case, it is the conjunction of all specification conjuncts that are state predicates.

*Next* The specification's next-state action. It is specified by a **NEXT** statement or a **SPECIFICATION** statement. In the latter case, it is the action  $N$  such that there is a specification conjunct of the form  $\square[N]_v$ . There must not be more than one such conjunct.

*Temporal* The conjunction of every specification conjunct that is neither a state predicate nor a box-action formula. It is usually the specification's liveness condition.

*Invariant* The conjunction of every state predicate  $I$  that is either named by an **INVARIANT** statement or for which some property conjunct equals  $\square I$ .

*ImpliedInit* The conjunction of every property conjunct that is a state predicate.

*ImpliedAction* The conjunction of every action  $[A]_v$  such that some property conjunct equals  $\square[A]_v$ .

*ImpliedTemporal* The conjunction of every property conjunct that is a simple temporal formula, but is not of the form  $\square I$ , where  $I$  is a state predicate.

*Constraint* The conjunction of all state predicates named by CONSTRAINT statements.

*ActionConstraint* The conjunction of all actions named by ACTION-CONSTRAINT statements. An action constraint is similar to an ordinary constraint, except it eliminates possible transitions rather than states. An ordinary constraint  $P$  is equivalent to the action constraint  $P'$ .

### 14.3.1 Model-Checking Mode

TLC keeps two data structures: a directed graph  $\mathcal{G}$  whose nodes are states, and a queue (a sequence)  $\mathcal{U}$  of states. A *state in  $\mathcal{G}$*  means a state that is a node of the graph  $\mathcal{G}$ . The graph  $\mathcal{G}$  is the part of the state reachability graph that TLC has found so far, and  $\mathcal{U}$  contains all states in  $\mathcal{G}$  whose successors TLC has not yet computed. TLC's computation maintains the following invariants:

- The states of  $\mathcal{G}$  satisfy the *Constraint* predicate.
- For every state  $s$  in  $\mathcal{G}$ , the edge from  $s$  to  $s$  is in  $\mathcal{G}$ .
- If there is an edge in  $\mathcal{G}$  from state  $s$  to a different state  $t$ , then  $t$  is a successor state of  $s$  that satisfies the action constraint. In other words, the step  $s \rightarrow t$  satisfies  $Next \wedge ActionConstraint$ .
- Each state  $s$  of  $\mathcal{G}$  is reachable from an initial state (one that satisfies the *Init* predicate) by a path in  $\mathcal{G}$ .
- $\mathcal{U}$  is a sequence of distinct states that are nodes in  $\mathcal{G}$ .
- For every state  $s$  in  $\mathcal{G}$  that is not in  $\mathcal{U}$ , and for every state  $t$  satisfying *Constraint* such that the step  $s \rightarrow t$  satisfies  $Next \wedge ActionConstraint$ , the state  $t$  and the edge from  $s$  to  $t$  are in  $\mathcal{G}$ .

TLC executes the following algorithm, starting with  $\mathcal{G}$  and  $\mathcal{U}$  empty:

1. Check that every *ASSUME* in the specification is satisfied by the values assigned to the constant parameters.
2. Compute the set of initial states by evaluating the initial predicate *Init*, as described above in Section 14.2.6. For each initial state  $s$  found:

- (a) Evaluate the predicates *Invariant* and *ImpliedInit* in state  $s$ ; report an error and stop if either is false.
  - (b) If the predicate *Constraint* is true in state  $s$ , then add  $s$  to the queue  $\mathcal{U}$  and add node  $s$  and edge  $s \rightarrow s$  to the graph  $\mathcal{G}$ .
3. While  $\mathcal{U}$  is nonempty, do the following:
- (a) Remove the first state from  $\mathcal{U}$  and let  $s$  be that state.
  - (b) Find the set  $T$  of all successor states of  $s$  by evaluating the next-state action starting from  $s$ , as described above in Section 14.2.6.
  - (c) If  $T$  is empty and the *deadlock* option is *not* selected, then report a deadlock error and stop.
  - (d) For each state  $t$  in  $T$ , do the following:
    - i. If *Invariant* is false in state  $t$  or *ImpliedAction* is false for the step  $s \rightarrow t$ , then report an error and stop.
    - ii. If the predicate *Constraint* is true in state  $t$  and the step  $s \rightarrow t$  satisfies *ActionConstraint*, then
      - A. If  $t$  is not in  $\mathcal{G}$ , then add it to the tail of  $\mathcal{U}$  and add the node  $t$  and the edge  $t \rightarrow t$  to  $\mathcal{G}$ .
      - B. Add the edge  $s \rightarrow t$  to  $\mathcal{G}$ .

TLC can use multiple threads, and steps 3(b)–(d) may be performed concurrently by different threads for different states  $s$ . See the description of the *workers* option on page 253 below.

If formula *ImpliedTemporal* is not equal to TRUE, then whenever it adds an edge  $s \rightarrow t$  in the procedure above, TLC evaluates all the predicates and actions that appear in formulas *Temporal* and *ImpliedTemporal* for the step  $s \rightarrow t$ . (It does this when adding any edge, including the self-loops  $s \rightarrow s$  and  $t \rightarrow t$  in steps 2(b) and 3(d)ii.A.)

Periodically during the computation of  $\mathcal{G}$ , and when it has finished computing  $\mathcal{G}$ , TLC checks the *ImpliedTemporal* property as follows. Let  $\mathcal{T}$  be the set consisting of every behavior  $\tau$  that is the sequence of states in an infinite path in  $\mathcal{G}$  starting with an initial state. (For example,  $\mathcal{T}$  contains the path  $s \rightarrow s \rightarrow s \rightarrow \dots$  for every initial state  $s$  in  $\mathcal{G}$ .) Note that every behavior in  $\mathcal{T}$  satisfies  $Init \wedge \square[Next]_{vars}$ . TLC checks that every behavior in  $\mathcal{T}$  also satisfies the formula  $Temporal \Rightarrow ImpliedTemporal$ . (This is conceptually what happens; TLC does not actually check each behavior separately.) See Section 14.3.5 on page 247 below for a discussion of why TLC’s checking of the *ImpliedTemporal* property may not do what you expect.

The computation of  $\mathcal{G}$  terminates only if the set of reachable states is finite. Otherwise, TLC will run forever—that is, until it runs out of resources or is stopped.

See page 226 for the definition of *reachable state*.

TLC does not always perform all three of the steps described above. It does step 2 only for a non-constant module, in which case the configuration file must specify an *Init* formula. TLC does step 3 only if the configuration file specifies a *Next* formula, which it must do if it specifies an *Invariant*, *ImpliedAction*, or *ImpliedTemporal* formula.

### 14.3.2 Simulation Mode

In simulation mode, TLC repeatedly constructs and checks individual behaviors of a fixed maximum length. The maximum length can be specified with the *depth* option, as described on page 251 below. (Its default value is 100 states.) In simulation mode, TLC runs until you stop it.

To create and check a behavior, TLC uses the procedure described above for constructing the graph  $\mathcal{G}$ —except with the following difference. After computing the set of initial states, and after computing the set  $T$  of successors for a state  $s$ , TLC randomly chooses an element of that set. If the element does not satisfy the constraint, then the computation of  $\mathcal{G}$  stops. Otherwise, TLC puts only that state in  $\mathcal{G}$  and  $\mathcal{U}$ , and checks the *Invariant* and the *ImpliedInit* or the *ImpliedAction* formula for it. (The queue  $\mathcal{U}$  isn't actually maintained, since it would never contain more than a single element.) The construction of  $\mathcal{G}$  stops, and the formula  $\text{Temporal} \Rightarrow \text{ImpliedTemporal}$  is checked, when the specified maximum number of states have been generated. TLC then repeats the procedure, starting with  $\mathcal{G}$  and  $\mathcal{U}$  empty.

TLC's choices are not strictly random, but are generated using a pseudo-random number generator from a randomly chosen seed. The seed and another value called the *aril* are printed if TLC finds an error. As described in Section 14.5.1 below, using the *key* and *aril* options, you can get TLC to generate the behavior that displayed the error.

### 14.3.3 Views and Fingerprints

In the description above of how TLC checks properties, I wrote that the nodes of the graph  $\mathcal{G}$  are states. That is not quite correct. The nodes of  $\mathcal{G}$  are values of a state function called the *view*. TLC's default view is the tuple of all declared variables, whose value determines the state. However, you can specify that the view should be some other state function *myview* by putting the statement

```
VIEW myview
```

in the configuration file, where *myview* is an identifier that is either defined or else declared to be a variable.

When TLC computes initial states, it puts their views rather than the states themselves in  $\mathcal{G}$ . (The view of a state  $s$  is the value of the `VIEW` state function in

Remember that we are using the term *state* informally to mean an assignment of values to declared variables, rather than to all variables.

state  $s$ .) If there are multiple initial states with the same view, only one of them is put in the queue  $\mathcal{U}$ . Instead of inserting an edge from a state  $s$  to a state  $t$ , TLC inserts the edge from the view of  $s$  to the view of  $t$ . In step 3(d)ii.A in the algorithm above, TLC checks if the view of  $t$  is in  $\mathcal{G}$ .

When using a view other than the default one, TLC may stop before it has found all reachable states. For the states it does find, it correctly performs safety checks—that is, the *Invariant*, *ImpliedInit*, and *ImpliedAction* checks. Moreover, it prints out a correct counterexample (a finite sequence of states) if it finds an error in one of those properties. However, it may incorrectly check the *ImpliedTemporal* property. Because the graph  $\mathcal{G}$  that TLC is constructing is not the actual reachability graph, it may report an error in the *ImpliedTemporal* property when none exists, printing out a bogus counterexample.

Specifying a nonstandard view can cause TLC not to check many states. You should do it when there is no need to check different states that have the same view. The most likely alternate view is a tuple consisting of some, but not all, declared variables. For example, you may have added one or more variables to help debug the specification. Using the tuple of the original variables as the view lets you add debugging variables without increasing the number of states that TLC must explore. If the properties being checked do not mention the debugging variables, then TLC will find all reachable states of the original specification and will correctly check all properties.

In the actual implementation, the nodes of the graph  $\mathcal{G}$  are not the views of states, but *fingerprints* of those views. A TLC fingerprint is a 64-bit number generated by a “hashing” function. Ideally, the probability that two different views have the same fingerprint is  $2^{-64}$ , which is a very small number. However, it is possible for a *collision* to occur, meaning that TLC mistakenly thinks that two different views are the same because they have the same fingerprint. If this happens, TLC will not explore all the states that it should. In particular, with the default view, TLC will report that it has checked all reachable states when it hasn’t.

When it terminates, TLC prints out two estimates of the probability that a fingerprint collision occurred. The first is based on the assumption that the probability of two different views having the same fingerprint is  $2^{-64}$ . (Under this assumption, if TLC generated  $n$  views with  $m$  distinct fingerprints, then the probability of a collision is about  $m * (n - m) * 2^{-64}$ .) However, the process of generating states is highly nonrandom, and no known fingerprinting scheme can guarantee that the probability of any two distinct states generated by TLC having the same fingerprint is actually  $2^{-64}$ . So, TLC also prints an empirical estimate of the probability that a collision occurred. It is based on the observation that, if there was a collision, then it is likely that there was also a “near miss”. The estimate is the maximum value of  $1/|f_1 - f_2|$  over all pairs  $\langle f_1, f_2 \rangle$  of distinct fingerprints generated by TLC. In practice, the probability of collision turns out to be very small unless TLC is generating billions of distinct states.

Views and fingerprinting apply only to model-checking mode. In simulation mode, TLC ignores any `VIEW` statement.

### 14.3.4 Taking Advantage of Symmetry

The memory specifications of Chapter 5 are symmetric in the set *Proc* of processors. Intuitively, this means that permuting the processors doesn't change whether or not a behavior satisfies a specification. To define symmetry more precisely, we first need some definitions.

A *permutation* of a finite set *S* is a function whose domain and range both equal *S*. In other words,  $\pi$  is a permutation of *S* iff

$$(S = \text{DOMAIN } \pi) \wedge (\forall w \in S : \exists v \in S : \pi[v] = w)$$

A *permutation* is a function that is a permutation of its (finite) domain. If  $\pi$  is a permutation of a set *S* of values and *s* is a state, let  $s^\pi$  be the state obtained from *s* by replacing each value *v* in *S* with  $\pi[v]$ . To see what  $s^\pi$  means, let's take as an example the permutation  $\pi$  of  $\{\text{"a"}, \text{"b"}, \text{"c"}\}$  such that  $\pi[\text{"a"}] = \text{"b"}$ ,  $\pi[\text{"b"}] = \text{"c"}$ , and  $\pi[\text{"c"}] = \text{"a"}$ . Suppose that, in state *s*, the values of the variables *x* and *y* are

$$\begin{aligned} x &= \langle \text{"b"}, \text{"c"}, \text{"d"} \rangle \\ y &= [i \in \{\text{"a"}, \text{"b"}\} \mapsto \text{IF } i = \text{"a"} \text{ THEN } 7 \text{ ELSE } 42] \end{aligned}$$

Then in state  $s^\pi$ , the values of the variables *x* and *y* are

$$\begin{aligned} x &= \langle \text{"c"}, \text{"a"}, \text{"d"} \rangle \\ y &= [i \in \{\text{"b"}, \text{"c"}\} \mapsto \text{IF } i = \text{"b"} \text{ THEN } 7 \text{ ELSE } 42] \end{aligned}$$

This example should give you an intuitive idea of what  $s^\pi$  means; I won't try to define it rigorously. If  $\sigma$  is the behavior  $s_1, s_2, \dots$ , let  $\sigma^\pi$  be the behavior  $s_1^\pi, s_2^\pi, \dots$ .

We can now define what symmetry means. A specification *Spec* is *symmetric with respect to* a permutation  $\pi$  iff the following condition holds: for any behavior  $\sigma$ , formula *Spec* is satisfied by  $\sigma$  iff it is satisfied by  $\sigma^\pi$ .

The memory specifications of Chapter 5 are symmetric with respect to any permutation of *Proc*. This means that there is no need for TLC to check a behavior  $\sigma$  if it has already checked the behavior  $\sigma^\pi$  for some permutation  $\pi$  of *Proc*. (Any error revealed by  $\sigma$  would also be revealed by  $\sigma^\pi$ .) We can tell TLC to take advantage of this symmetry by putting the following statement in the configuration file:

`SYMMETRY Perms`

where *Perms* is defined in the module to equal *Permutations(Proc)*, the set of all permutations of *Proc*. (The *Permutations* operator is defined in the *TLC*

module, described in Section 14.4 below.) This **SYMMETRY** statement causes TLC to modify the algorithm described on pages 241–242 so that it never adds a state  $s$  to its queue  $\mathcal{U}$  of unexamined states and to its state graph  $\mathcal{G}$  if  $\mathcal{G}$  already contains the state  $s^\pi$ , for some permutation  $\pi$  of *Proc*. If there are  $n$  processes, this reduces the number of states that TLC examines by a factor of  $n!$ .

The memory specifications of Chapter 5 are also symmetric with respect to any permutation of the set *Adr* of memory addresses. To take advantage of this symmetry as well as the symmetry with respect to permutations of processors, we define the symmetry set (the set specified by the **SYMMETRY** statement) to equal

$$\text{Permutations}(\text{Proc}) \cup \text{Permutations}(\text{Adr})$$

In general, the **SYMMETRY** statement can specify an arbitrary symmetry set  $\Pi$ , each element of which is a permutation of a set of model values. More precisely, each element  $\pi$  in  $\Pi$  must be a permutation such that all the elements of  $\text{DOMAIN } \pi$  are assigned model values by the configuration file's **CONSTANT** statement. (If the configuration has no **SYMMETRY** statement, we take the symmetry set  $\Pi$  to be the empty set.)

To explain what TLC does when given an arbitrary symmetry set  $\Pi$ , I need a few more definitions. If  $\tau$  is a sequence  $\langle \pi_1, \dots, \pi_n \rangle$  of permutations in  $\Pi$ , let  $s^\tau$  equal  $(\dots((s^{\pi_1})^{\pi_2})\dots)^{\pi_n}$ . (If  $\tau$  is the empty sequence, then  $s^\tau$  is defined to equal  $s$ .) Define the *equivalence class*  $\hat{s}$  of a state  $s$  to be the set of states  $s^\tau$  for all sequences  $\tau$  of permutations in  $\Pi$ . For any state  $s$ , TLC keeps only a single element of  $\hat{s}$  in  $\mathcal{U}$  and  $\mathcal{G}$ . This is accomplished by the following modifications to the algorithm on pages 241–242. In step 2(b), TLC adds the state  $s$  to  $\mathcal{U}$  and  $\mathcal{G}$  only if  $\mathcal{U}$  and  $\mathcal{G}$  do not already contain a state in  $\hat{s}$ . Step 3(d)ii is changed to

- A. If no element in  $\hat{t}$  is in  $\mathcal{G}$ , then add  $t$  to the tail of  $\mathcal{U}$  and add the node  $t$  and the edge  $t \rightarrow t$  to  $\mathcal{G}$ .
- B. Add the edge  $s \rightarrow tt$  to  $\mathcal{G}$ , where  $tt$  is the unique element of  $\hat{t}$  that is (now) in  $\mathcal{G}$ .

When a **VIEW** statement appears in the configuration file, these changes are modified as described in Section 14.3.3 above so that views rather than states are put in  $\mathcal{G}$ .

If the specification and the properties being checked are, indeed, symmetric with respect to all permutations in the symmetry set, then TLC's *Invariant*, *ImpliedInit*, and *ImpliedAction* checking will find and correctly report any error that they would have found had the **SYMMETRY** statement been omitted. However, TLC may perform *ImpliedTemporal* checking incorrectly—it may miss errors, report an error that doesn't exist, or report a real error with an incorrect counterexample. So, you should do *ImpliedTemporal* checking when using a **SYMMETRY** statement only if you understand exactly what TLC is doing.

If the specification and properties are not symmetric with respect to all permutations in the symmetry set, then TLC may be unable to print an error trace if it does find an error. In that case, it will print the error message

**Failed to recover the state from its fingerprint.**

The symmetry set is used only in model-checking mode. TLC ignores it in simulation mode.

### 14.3.5 Limitations of Liveness Checking

If a specification violates a safety property, then there is a finite behavior that displays the violation. That behavior can be generated with a finite model. It is therefore, in principle, possible to discover the violation with TLC. It may be impossible to discover a violation of a liveness property with any finite model. To see why, consider the following simple specification *EvenSpec* that starts with  $x$  equal to zero and repeatedly increments it by 2:

$$\text{EvenSpec} \triangleq (x = 0) \wedge \square[x' = x + 2]_x \wedge \text{WF}_x(x' = x + 2)$$

Safety properties  
were defined on  
page 87.

Obviously,  $x$  never equals 1 in any behavior satisfying *EvenSpec*. So, *EvenSpec* does not satisfy the liveness property  $\diamond(x = 1)$ . Suppose we ask TLC to check if *EvenSpec* implies  $\diamond(x = 1)$ . To get TLC to terminate, we must provide a constraint that limits it to generating a finite number of reachable states. All the infinite behaviors satisfying  $(x = 0) \wedge \square[x' = x + 2]_x$  that TLC generates will then end in an infinite number of stuttering steps. In any such behavior, action  $x' = x + 2$  is always enabled, but only a finite number of  $x' = x + 2$  steps occur, so  $\text{WF}_x(x' = x + 2)$  is false. TLC will therefore not report an error because the formula

$$\text{WF}_x(x' = x + 2) \Rightarrow \diamond(x = 1)$$

is satisfied by all the infinite behaviors it generates.

When doing temporal checking, make sure that your model will permit infinite behaviors that satisfy the specification's liveness condition. For example, consider the finite model of the alternating bit protocol specification defined by the configuration file of Figure 14.3 on page 227. You should convince yourself that it allows infinite behaviors that satisfy formula *ABFairness*.

It's a good idea to verify that TLC is performing the liveness checking you expect. Have it check a liveness property that the specification does not satisfy and make sure it reports an error.

MODULE *TLC*LOCAL INSTANCE *Naturals*The keyword LOCAL means that definitions from the instantiated module are not obtained by a module that extends *TLC*.LOCAL INSTANCE *Sequences*

## OPERATORS FOR DEBUGGING

 $Print(out, val) \triangleq val$  Causes TLC to print the values *out* and *val*. $Assert(val, out) \triangleq \text{IF } val = \text{TRUE} \text{ THEN TRUE}$  Causes TLC to report an error  
 $\text{ELSE CHOOSE } v : \text{TRUE}$  and print *out* if *val* is not true. $JavaTime \triangleq \text{CHOOSE } n : n \in \text{Nat}$  Causes TLC to print the current time, in milliseconds elapsed  
since 00:00 on 1 Jan 1970 UT, modulo  $2^{31}$ .

## OPERATORS FOR REPRESENTING FUNCTIONS AND SETS OF PERMUTATIONS

 $d :> e \triangleq [x \in \{d\} \mapsto e]$ The function *f* with domain  $\{d_1, \dots, d_n\}$   
such that  $f[d_i] = e_i$ , for  $i = 1, \dots, n$  can be  
written  
 $d_1 :> e_1 @\dots @\dots @\dots @\dots d_n :> e_n$  $f @\dots @\dots g \triangleq [x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto$  $\text{IF } x \in \text{DOMAIN } f \text{ THEN } f[x] \text{ ELSE } g[x]]$  $Permutations(S) \triangleq \{f \in [S \rightarrow S] : \forall w \in S : \exists v \in S : f[v] = w\}$  The set of permutations of *S*.

## AN OPERATOR FOR SORTING

 $SortSeq(s, \prec \dots) \triangleq$  The result of sorting sequence *s* according to the ordering  $\prec$ .LET *Perm*  $\triangleq$  CHOOSE *p*  $\in$  *Permutations*(1 .. *Len(s)*) : $\forall i, j \in 1 \dots \text{Len}(s) : (i < j) \Rightarrow (s[p[i]] \prec s[p[j]]) \vee (s[p[i]] = s[p[j]])$ IN  $[i \in 1 \dots \text{Len}(s) \mapsto s[\text{Perm}[i]]]$ Figure 14.5: The standard module *TLC*.14.4 The *TLC* Module

The standard *TLC* module, in Figure 14.5 on this page, defines operators that are handy when using TLC. The module on which you run TLC usually EXTENDS the *TLC* module, which is overridden by its Java implementation.

Module *TLC* begins with the statementModule overriding  
is explained above  
in Section 14.2.5.LOCAL INSTANCE *Naturals*

As explained on page 171, this is like an EXTENDS statement, except that the definitions included from the *Naturals* module are not obtained by any other module that extends or instantiates module *TLC*. Similarly, the next statement locally instantiates the *Sequences* module.

Module *TLC* next defines three operators *Print*, *Assert*, and *JavaTime*. They are of no use except in running TLC, when they can help you track down problems.

The operator *Print* is defined so that  $\text{Print}(\text{out}, \text{val})$  equals  $\text{val}$ . But when TLC evaluates this expression, it prints the values of *out* and *val*. You can add *Print* expressions to a specification to help locate an error. For example, if your specification contains

$$\begin{aligned} & \wedge \text{Print}(\text{"a"}, \text{TRUE}) \\ & \wedge P \\ & \wedge \text{Print}(\text{"b"}, \text{TRUE}) \end{aligned}$$

and TLC prints the "a" but not the "b" before reporting an error, then the error happened while TLC was evaluating *P*. If you know where the error is but don't know why it's occurring, you can add *Print* expressions to give you more information about what values TLC has computed.

To understand what gets printed when, you must know how TLC evaluates expressions, which is explained above in Sections 14.2 and 14.3. TLC usually evaluates an expression many times, so inserting a *Print* expression in the specification can produce a lot of output. One way to limit the amount of output is to put the *Print* expression inside an IF/THEN expression, so it is executed only in interesting cases.

The *TLC* module next defines the operator *Assert* so  $\text{Assert}(\text{val}, \text{out})$  equals TRUE if *val* equals TRUE. If *val* does not equal TRUE, evaluating  $\text{Assert}(\text{val}, \text{out})$  causes TLC to print the value of *out* and to halt. (In this case, the value of  $\text{Assert}(\text{val}, \text{out})$  is irrelevant.)

Next, the operator *JavaTime* is defined to equal an arbitrary natural number. However, TLC does not obey the definition of *JavaTime* when evaluating it. Instead, evaluating *JavaTime* yields the time at which the evaluation takes place, measured in milliseconds elapsed since 00:00 Universal Time on 1 January 1970, modulo  $2^{31}$ . If TLC is generating states slowly, using the *JavaTime* operator in conjunction with *Print* expressions can help you understand why. If TLC is spending too much time evaluating an operator, you may be able to replace the operator's definition with an equivalent one that TLC can evaluate more efficiently. (See Section 14.2.3 on page 234.)

The TLC module next defines the operators  $:>$  and  $@@$  so that the expression

$$d_1 :> e_1 @@ \dots @@ d_n :> e_n$$

is the function *f* with domain  $\{d_1, \dots, d_n\}$  such that  $f[d_i] = e_i$ , for  $i = 1, \dots, n$ . For example, the sequence  $\langle \text{"ab"}, \text{"cd"} \rangle$ , which is a function with domain  $\{1, 2\}$ , can be written as

$$1 :> \text{"ab"} @@ 2 :> \text{"cd"}$$

TLC uses these operators to represent function values that it prints when evaluating a *Print* expression or reporting an error. However, it usually prints values the way they appear in the specification, so it usually prints a sequence as a sequence, not in terms of the  $:>$  and  $@@$  operators.

Next comes the definition of  $\text{Permutations}(S)$  to be the set of all permutations of  $S$ , if  $S$  is a finite set. The  $\text{Permutations}$  operator can be used to specify a set of permutations for the **SYMMETRY** statement described in Section 14.3.4 above. More complicated symmetries can be expressed by defining a set  $\{\pi_1, \dots, \pi_n\}$  of permutations, where each  $\pi_i$  is written as an explicit function using the  $:>$  and  $@@$  operators. For example, consider a specification of a memory system in which each address is in some way associated with a processor. The specification would be symmetric under two kinds of permutations: ones that permute addresses associated with the same processor, and ones that permute the processors along with their associated addresses. Suppose we tell TLC to use two processors and four addresses, where addresses  $a11$  and  $a12$  are associated with processor  $p1$  and addresses  $a21$  and  $a22$  are associated with processor  $p2$ . We can get TLC to take advantage of the symmetries by giving it the following set of permutations as the symmetry set:

$$\begin{aligned} \text{Permutations}(\{a11, a12\}) \cup \{p1:>p2 @@ p2:>p1 \\ @@ a11:>a21 @@ a21:>a11 \\ @@ a12:>a22 @@ a22:>a12\} \end{aligned}$$

The permutation  $p1:>p2 @@ \dots @@ a22:>a12$  interchanges the processors and their associated addresses. The permutation that just interchanges  $a21$  and  $a22$  need not be specified explicitly because it is obtained by interchanging the processors, interchanging  $a11$  and  $a12$ , and interchanging the processors again.

The TLC module ends by defining the operator  $\text{SortSeq}$ , which can be used to replace operator definitions with ones that TLC can evaluate more efficiently. If  $s$  is a finite sequence and  $\prec$  is a total ordering relation on its elements, then  $\text{SortSeq}(s, \prec)$  is the sequence obtained from  $s$  by sorting its elements according to  $\prec$ . For example,  $\text{SortSeq}(\langle 3, 1, 3, 8 \rangle, >)$  equals  $\langle 8, 3, 3, 1 \rangle$ . The Java implementation of  $\text{SortSeq}$  allows TLC to evaluate it more efficiently than a user-defined sorting operator. For example, here's how we can use  $\text{SortSeq}$  to define an operator  $\text{FastSort}$  to replace the  $\text{Sort}$  operator defined on page 235.

$$\begin{aligned} \text{FastSort}(S) &\triangleq \\ \text{LET } \text{MakeSeq}[SS \in \text{SUBSET } S] &\triangleq \\ \text{IF } SS = \{\} \text{ THEN } \langle \rangle & \\ \text{ELSE } \text{LET } ss \triangleq \text{CHOOSE } ss \in SS : \text{TRUE} & \\ \text{IN } \text{Append}(\text{MakeSeq}[SS \setminus \{ss\}], ss) & \\ \text{IN } \text{SortSeq}(\text{MakeSeq}[S], <) & \end{aligned}$$

## 14.5 How to Use TLC

### 14.5.1 Running TLC

Exactly how you run TLC depends on what operating system you are using and how it is configured. You will probably type a command of the form

*program-name* *options* *spec-file*

where

*program-name* is specific to your system. It might be `java tlatk.TLC`.

*spec-file* is the name of the file containing the TLA<sup>+</sup> specification. Each TLA<sup>+</sup> module named *M* that appears in the specification must be in a separate file named *M.tla*. The extension `.tla` may be omitted from *spec-file*.

*options* is a sequence consisting of zero or more of the following options:

**-deadlock**

Tells TLC not to check for deadlock. Unless this option is specified, TLC will stop if it finds a deadlock—that is, a reachable state with no successor state.

**-simulate**

Tells TLC to run in simulation mode, generating randomly chosen behaviors, instead of generating all reachable states. (See Section 14.3.2 above.)

**-depth num**

This option causes TLC to generate behaviors of length at most *num* in simulation mode. Without this option, TLC will generate runs of length at most 100. This option is meaningful only when the *simulate* option is used.

**-seed num**

In simulation mode, the behaviors generated by TLC are determined by the initial seed given to a pseudorandom number generator. Normally, the seed is generated randomly. This option causes TLC to let the seed be *num*, which must be an integer from  $-2^{63}$  to  $2^{63} - 1$ . Running TLC twice in simulation mode with the same seed and *aril* (see the *aril* option below) will produce identical results. This option is meaningful only when using the *simulate* option.

**-aril num**

This option causes TLC to use *num* as the *aril* in simulation mode. The *aril* is a modifier of the initial seed. When TLC finds an error in simulation mode, it prints out both the initial seed and an *aril*

number. Using this initial seed and `aril` will cause the first trace generated to be that error trace. Adding `Print` expressions will usually not change the order in which TLC generates traces. So, if the trace doesn't tell you what went wrong, you can try running TLC again on just that trace to print out additional information.

**`-coverage num`**

This option causes TLC to print “coverage” information every *num* minutes and at the end of its execution. For every action conjunct that “assigns a value” to a variable, TLC prints the number of times that conjunct has actually been used in constructing a new state. The values it prints may not be accurate, but their magnitude can provide useful information. In particular, a value of 0 indicates part of the next-state action that was never “executed”. This might indicate an error in the specification, or it might mean that the model TLC is checking is too small to exercise that part of the action.

**`-recover run_id`**

This option causes TLC to start executing the specification not from the beginning, but from where it left off at the last checkpoint. When TLC takes a checkpoint, it prints the run identifier. (That identifier is the same throughout an execution of TLC.) The value of *run\_id* should be that run identifier.

**`-cleanup`**

TLC creates a number of files when it runs. When it completes, it erases all of them. If TLC finds an error, or if you stop it before it finishes, TLC can leave some large files around. The *cleanup* option causes TLC to delete all files created by previous runs. Do not use this option if you are currently running another copy of TLC in the same directory; if you do, it can cause the other copy to fail.

**`-difftrace num`**

When TLC finds an error, it prints an error trace. Normally, that trace is printed as a sequence of complete states, where a state lists the values of all declared variables. The *difftrace* option causes TLC to print an abridged version of each state, listing only the variables whose values are different than in the preceding state. This makes it easier to see what is happening in each step, but harder to find the complete state.

**`-terse`**

Normally, TLC completely expands values that appear in error messages or in the output from evaluating `Print` expressions. The *terse* option causes TLC instead to print partially evaluated, shorter versions of these values.

**-workers** *num*

Steps 3(b)–(d) of the TLC execution algorithm described on pages 241–242 can be speeded up on a multiprocessor computer by the use of multiple threads. This option causes TLC to use *num* threads when finding reachable states. There is no reason to use more threads than there are actual processors on your computer. If the option is omitted, TLC uses a single thread.

**-config** *config\_file*

Specifies that the configuration file is named *config\_file*, which must be a file with extension `.cfg`. The extension `.cfg` may be omitted from *config\_file*. If this option is omitted, the configuration file is assumed to have the same name as *spec\_file*, except with the extension `.cfg`.

**-nowarning**

There are TLA<sup>+</sup> expressions that are legal but are sufficiently unlikely that their presence probably indicates an error. For example, the expression  $[f \text{ EXCEPT } ![v] = e]$  is probably incorrect if *v* is not an element of the domain of *f*. (In this case, the expression just equals *f*.) TLC normally issues a warning when it encounters such an unlikely expression; this option suppresses these warnings.

## 14.5.2 Debugging a Specification

When you write a specification, it usually contains errors. The purpose of running TLC is to find as many of those errors as possible. We hope an error in the specification will cause TLC to report an error. The challenge of debugging is to find the error in the specification that caused the error reported by TLC. Before addressing this challenge, let's first examine TLC's output when it finds no error.

### TLC's Normal Output

When you run TLC, the first thing it prints is the version number and creation date:

TLC Version 2.12 of 26 May 2003

Always include this information when reporting any problems with TLC. Next, TLC describes the mode in which it's being run. The possibilities are

**Model-checking**

in which it is exhaustively checking all reachable states, or

TLC's messages  
may differ in for-  
mat from the ones  
described here.

```
Running Random Simulation with seed 1901803014088851111.
```

in which it is running in simulation mode, using the indicated seed. (Seeds are described on pages 251–252.) Let’s suppose it’s running in model-checking mode. If you asked TLC to do liveness checking, it will now print something like

```
Implied-temporal checking--relative complexity = 8.
```

The time TLC takes for liveness checking is approximately proportional to the relative complexity. Even with a relative complexity of 1, checking liveness takes longer than checking safety. So, if the relative complexity is not small, TLC will probably take a very long time to complete, unless the model is very small. In simulation mode, a large complexity means that TLC will not be able to simulate very many behaviors. The relative complexity depends on the number of terms and the size of sets being quantified over in the temporal formulas.

TLC next prints a message like

```
Finished computing initial states:
4 states generated, with 2 of them distinct.
```

This indicates that, when evaluating the initial predicate, TLC generated 4 states, among which there were 2 distinct ones. TLC then prints one or more messages such as

```
Progress(9): 2846 states generated, 984 distinct states
found. 856 states left on queue.
```

This message indicates that TLC has thus far constructed a state graph  $\mathcal{G}$  of diameter<sup>2</sup> 9, that it has generated and examined 2846 states, finding 984 distinct ones, and that the queue  $\mathcal{U}$  of unexplored states contains 856 states. After running for a while, TLC generates these progress reports about once every five minutes. For most specifications, the number of states on the queue increases monotonically at the beginning of the execution and decreases monotonically at the end. The progress reports therefore provide a useful guide to how much longer the execution is likely to take.

$\mathcal{G}$  and  $\mathcal{U}$  are described in Section 14.3.1 on page 241.

When TLC successfully completes, it prints

```
Model checking completed. No error has been found.
```

It then prints something like

---

<sup>2</sup>The diameter of  $\mathcal{G}$  is the smallest number  $d$  such that every state in  $\mathcal{G}$  can be reached from an initial state by a path containing at most  $d$  states. It is the depth TLC has reached in its breadth-first exploration of the set of states. When using multiple threads (specified with the *workers* option), the diameter TLC reports may not be quite correct.

```
Estimates of the probability that TLC did not check all
reachable states because two distinct states had the same
fingerprint:
```

```
calculated (optimistic): .000003
based on the actual fingerprints: .00007
```

As explained on page 244, these are TLC's two estimates of the probability of a fingerprint collision. Finally, TLC prints a message like

```
2846 states generated, 984 distinct states found,
0 states left on queue.
The state graph has diameter 15.
```

with the total number of states and the diameter of the state graph.

While TLC is running, it may also print a message such as

```
-- Checkpointing run states/99-05-20-15-47-55 completed
```

This indicates that it has written a checkpoint that you can use to restart TLC in the event of a computer failure. (As explained in Section 14.5.3 on page 260, checkpoints have other uses as well.) The run identifier

```
states/99-05-20-15-47-55
```

is used with the *recover* option to restart TLC from where the checkpoint was taken. If only part of this message was printed—for example, because your computer crashed while TLC was taking the checkpoint—there is a slight chance that all the checkpoints are corrupted and you must start TLC again from the beginning.

## Error Reports

The first problems you find in your specification will probably be syntax errors. TLC reports them with

```
ParseException in parseSpec:
```

followed by the error message generated by the Syntactic Analyzer. Chapter 12 explains how to interpret the analyzer's error messages. Running your specification through the analyzer as you write it will catch a lot of simple errors quickly.

As explained in Section 14.3.1 above, TLC executes three basic phases. In the first phase, it checks assumptions; in the second, it computes the initial states; and in the third, it generates the successor states of states on the queue  $\mathcal{U}$  of unexplored states. You can tell if it has entered the third phase by whether or not it has printed the “initial states computed” message.

TLC's most straightforward error report occurs when it finds that one of the properties it is checking does not hold. Suppose we introduce an error into our alternating bit specification (Figure 14.1 on pages 223 and 224) by replacing the first conjunct of the invariant *ABTypeInv* with

$\wedge \text{msgQ} \in \text{Seq}(\text{Data})$

TLC quickly finds the error and prints

**Invariant ABTypeInv is violated**

It next prints a minimal-length<sup>3</sup> behavior that leads to the state not satisfying the invariant:

The behavior up to this point is:

STATE 1: <Initial predicate>

```
 $\wedge \text{rBit} = 0$
 $\wedge \text{sBit} = 0$
 $\wedge \text{ackQ} = << >>$
 $\wedge \text{rcvd} = \text{d1}$
 $\wedge \text{sent} = \text{d1}$
 $\wedge \text{sAck} = 0$
 $\wedge \text{msgQ} = << >>$
```

Note that TLC indicates which part of the next-state action allows the step that produces each state.

STATE 2: <Action at line 66 in AlternatingBit>

```
 $\wedge \text{rBit} = 0$
 $\wedge \text{sBit} = 1$
 $\wedge \text{ackQ} = << >>$
 $\wedge \text{rcvd} = \text{d1}$
 $\wedge \text{sent} = \text{d1}$
 $\wedge \text{sAck} = 0$
 $\wedge \text{msgQ} = << << 1, \text{d1} >> >>$
```

TLC prints each state as a TLA<sup>+</sup> predicate that determines the state. When printing a state, TLC describes functions using the operators  $:>$  and  $@@$  defined in the TLC module. (See Section 14.4 on page 248.)

The hardest errors to locate are usually the ones detected when TLC is forced to evaluate an expression that it can't handle, or one that is "silly" because its value is not specified by the semantics of TLA<sup>+</sup>. As an example, let's introduce a typical "off-by-one" error into the alternating bit protocol by replacing the second conjunct in the definition of *Lose* with

$$\exists i \in 1 \dots \text{Len}(q) : \\ q' = [j \in 1 \dots (\text{Len}(q) - 1) \mapsto \text{IF } j < i \text{ THEN } q[j - 1] \\ \text{ELSE } q[j]]$$

<sup>3</sup>When using multiple threads, it is possible, though unlikely, for there to be a shorter behavior that also violates the invariant.

If  $q$  has length greater than 1, then this defines  $Lose(q)[1]$  to equal  $q[0]$ , which is a nonsensical value if  $q$  is a sequence. (The domain of a sequence  $q$  is the set  $1 \dots Len(q)$ , which does not contain 0.) Running TLC produces the error message

```
Error: Applying tuple
<< << 1, d1 >>, << 1, d1 >> >>
to integer 0 which is out of domain.
```

It then prints a behavior leading to the error. TLC finds the error when evaluating the next-state action to compute the successor states for some state  $s$ , and  $s$  is the last state in that behavior. Had the error occurred when evaluating the invariant or the implied-action, TLC would have been evaluating it on the last state or step of the behavior.

Finally, TLC prints the location of the error:

```
The error occurred when TLC was evaluating the nested
expressions at the following positions:
0. Line 57, column 7 to line 59, column 60 in AlternatingBit
1. Line 58, column 55 to line 58, column 60 in AlternatingBit
```

The first position identifies the second conjunct of the definition of *Lose*; the second identifies the expression  $q[j - 1]$ . This tells you that the error occurred while TLC was evaluating  $q[j - 1]$ , which it was doing as part of the evaluation of the second conjunct of the definition of *Lose*. You must infer from the printed trace that it was evaluating the definition of *Lose* while evaluating the action *LoseMsg*. In general, TLC prints a tree of nested expressions—higher-level ones first. It seldom locates the error as precisely as you would like; often it just narrows it down to a conjunct or disjunct of a formula. You may need to insert *Print* expressions to locate the problem. See the discussion on page 259 for further advice on locating errors.

### 14.5.3 Hints on Using TLC Effectively

#### Start Small

The constraint and the assignment of values to the constant parameters define a model of the specification. How long it takes TLC to check a specification depends on the specification and the size of the model. Running on a 600MHz work station, TLC finds about 700 distinct reachable states per second for the alternating bit protocol specification. For some specifications, the time it takes TLC to generate a state grows with the size of the model; it can also increase as the generated states become more complicated. For some specifications run on larger models, TLC finds fewer than one reachable state per second.

You should always begin testing a specification with a tiny model, which TLC can check quickly. Let sets of processes and of data values have only one element; let queues be of length one. A specification that has not been tested probably has lots of errors. A small model will quickly catch most of the simple ones. When a very small model reveals no more errors, you can then run TLC with larger models to try to catch more subtle errors.

One way to figure out how large a model TLC can handle is to estimate the approximate number of reachable states as a function of the parameters. However, this can be hard. If you can't do it, increase the model size very gradually. The number of reachable states is typically an exponential function of the model's parameters; and the value of  $a^b$  grows very fast with increasing values of  $b$ .

Many systems have errors that will show up only on models too large for TLC to check exhaustively. After having TLC model check your specification on as large a model as your patience allows, you can run it in simulation mode on larger models. Random simulation is not an effective way to catch subtle errors, but it's worth trying; you might get lucky.

## Be Suspicious of Success

Section 14.3.5 on page 247 explains why you should be suspicious if TLC does not find a violation of a liveness property; the finite model may mask errors. You should also be suspicious if TLC finds no error when checking safety properties. It's very easy to satisfy a safety property by simply doing nothing. For example, suppose we forgot to include the *SndNewValue* action in the alternating bit protocol specification's next-state action. The sender would then never try to send any values. But the resulting specification would still satisfy the protocol's correctness condition, formula *ABCSpec* of module *ABCorrectness*. (The specification doesn't require that values must be sent.)

The *coverage* option described on page 252 provides one way to catch such problems. Another way is to make sure that TLC finds errors in properties that should be violated. For example, if the alternating bit protocol is sending messages, then the value of *sent* should change. You can verify that it does change by checking that TLC reports a violation of the property

$$\forall d \in Data : (sent = d) \Rightarrow \square(sent = d)$$

A good sanity check is to verify that TLC finds states that are reached only by performing a number of operations. For example, the caching memory specification of Section 5.6 should have reachable states in which a particular processor has both a read and two write operations in the *memQ* queue. Reaching such a state requires a processor to perform two writes followed by a read to an uncached address. We can verify that such a state is reachable by having TLC find a violation of an invariant declaring that there aren't a read and two writes for

the same processor in *memQ*. (Of course, this requires a model in which *memQ* can be large enough.) Another way to check that certain states are reached is by using the *Print* operator inside an IF/THEN expression in an invariant to print a message when a suitable state is reached.

## Let TLC Help You Figure Out What Went Wrong

When TLC reports that an invariant is violated, it may not be obvious what part of the invariant is false. If you give separate names to the conjuncts of your invariant and list them separately in the configuration file's INVARIANT statement, TLC will tell you which conjunct is false. However, it may be hard to see why even an individual conjunct is false. Instead of spending a lot of time trying to figure it out by yourself, it's easier to add *Print* expressions and let TLC tell you what's going wrong.

If you rerun TLC from the beginning with a lot of *Print* expressions, it will print output for every state it checks. Instead, you should start TLC from the state in which the invariant is false. Define a predicate, say *ErrorState*, that describes this state, and modify the configuration file to use *ErrorState* as the initial predicate. Writing the definition of *ErrorState* is easy—just copy the last state in TLC's error trace.<sup>4</sup>

You can use the same trick if any safety property is violated, or if TLC reports an error when evaluating the next-state action. For an error in a property of the form  $\square[A]_v$ , rerun TLC using the next-to-last state in the error trace as the initial predicate, and using the last state in the trace, with the variable names primed, as the next-state action. To find an error that occurs when evaluating the next-state action, use the last state in the error trace as the initial predicate. (In this case, TLC may find several successor states before reporting the error.)

If you have introduced model values in the configuration file, they will undoubtedly appear in the states printed by TLC. So, if you are to copy those states into the module, you will have to declare the model values as constant parameters and then assign to each of these parameters the model value of the same name. For example, the configuration file we used for the alternating bit protocol introduces model values *d1* and *d2*. So, we would add to module *MCAalternatingBit* the declaration

```
CONSTANTS d1, d2
```

and add to the CONSTANT statement of the configuration file the assignments

```
d1 = d1 d2 = d2
```

which assign to the constant parameters *d1* and *d2* the model values *d1* and *d2*, respectively.

---

<sup>4</sup>Defining *ErrorState* is not so easy if you use the *difftrace* option, which is a reason for not using that option.

## Don't Start Over After Every Error

After you've eliminated the errors that are easy to find, TLC may have to run for a long time before finding an error. Very often, it takes more than one try to fix an error properly. If you start TLC from the beginning after correcting an error, it may run for a long time only to report that you made a silly mistake in the correction. If the error was discovered when taking a step from a correct state, then it's a good idea to check your correction by starting TLC from that state. As explained above, you do this by defining a new initial predicate that equals the state printed by TLC.

Another way to avoid starting from scratch after an error is by using checkpoints. A checkpoint saves the current state graph  $\mathcal{G}$  and queue  $\mathcal{U}$  of unexplored states. It does not save any other information about the specification. You can restart TLC from a checkpoint even if you have changed the specification, as long as the specification's variables and the values that they can assume haven't changed. More precisely, you can restart from a checkpoint iff the view of any state computed before the checkpoint has not changed and the symmetry set is the same. When you correct an error that TLC found after running for a long time, you may want to use the *recover* option (page 252) to continue TLC from the last checkpoint instead of having it recheck all the states it has already checked.<sup>5</sup>

The view and symmetry set are defined in Sections 14.3.3 and 14.3.4, respectively.

## Check Everything You Can

Verify that your specification satisfies all the properties you think it should. For example, you shouldn't be content to check that the alternating bit protocol specification satisfies the higher-level specification *ABCSpec* of module *ABCorrectness*. You should also check lower-level properties that you expect it to satisfy. One such property, revealed by studying the algorithm, is that there should never be more than two different messages in the *msgQ* queue. So, we can check that the following predicate is an invariant:

$$\text{Cardinality}(\{ \text{msgQ}[i] : i \in 1 \dots \text{Len}(\text{msgQ}) \}) \leq 2$$

(We must add the definition of *Cardinality* to module *MCAlternatingBit* by adding *FiniteSets* to its EXTENDS statement.)

It's a good idea to check as many invariance properties as you can. If you think that some state predicate should be an invariant, let TLC test if it is. Discovering that the predicate isn't an invariant may not reveal an error, but it will probably teach you something about your specification.

---

<sup>5</sup>Some states in the graph  $\mathcal{G}$  may not be saved by a checkpoint; they will be rechecked when restarting from the checkpoint.

## Be Creative

Even if a specification seems to lie outside the realm of what it can handle, TLC may be able to help check it. For example, suppose a specification's next-state action has the form  $\exists n \in Nat : A(n)$ . TLC cannot evaluate quantification over an infinite set, so it apparently can't deal with this specification. However, we can enable TLC to evaluate the quantified formula by using the configuration file's `CONSTANT` statement to replace `Nat` with the finite set `0 .. n`, for some  $n$ . This replacement profoundly changes the specification's meaning. However, it might nonetheless allow TLC to reveal errors in the specification. Never forget that your objective in using TLC is not to verify that a specification is correct; it's to find errors.

Replacement is explained in Section 14.2.3.

## Use TLC as a TLA<sup>+</sup> Calculator

Misunderstanding some aspect of TLA<sup>+</sup> can lead to errors in your specification. Use TLC to check your understanding of TLA<sup>+</sup> by running it on small examples. TLC checks assumptions, so you can turn it into a TLA<sup>+</sup> calculator by having it check a module with no specification, only `ASSUME` statements. For example, if  $g$  equals

```
[f EXCEPT !(d) = e1, !(d) = e2]
```

what is the value of  $g[d]$ ? You can ask TLC by letting it check a module containing

```
ASSUME LET f \triangleq [i \in 1 .. 10 \mapsto 1]
 g \triangleq [f EXCEPT !(2) = 3, !(2) = 4]
 IN Print(g[2], TRUE)
```

You can have it verify that  $(F \Rightarrow G) \equiv (\neg F \vee G)$  is a tautology by checking

```
ASSUME $\forall F, G \in \text{BOOLEAN} : (F \Rightarrow G) \equiv (\neg F \vee G)$
```

TLC can even look for counterexamples to a conjecture. Can every set be written as the disjunction of two different sets? Check it for all subsets of  $1 .. 4$  with

```
ASSUME $\forall S \in \text{SUBSET}(1 .. 4) :$
 IF $\exists T, U \in \text{SUBSET}(1 .. 4) : (T \neq U) \wedge (S = T \cup U)$
 THEN TRUE
 ELSE Print(S, TRUE)
```

When TLC is run just to check assumptions, it may need no information from the configuration file. But you must provide a configuration file, even if that file is empty.

## 14.6 What TLC Doesn't Do

We would like TLC to generate all the behaviors that satisfy a specification. But no program can do this for an arbitrary specification. I have already mentioned some limitations of TLC. There are other limitations that you may stumble on. One of them is that the Java classes that override the *Naturals* and *Integers* modules handle only numbers in the interval  $-2^{31} \dots (2^{31} - 1)$ ; TLC reports an error if any computation generates a value outside this interval.

TLC can't generate all behaviors satisfying an arbitrary specification, but it might achieve the easier goal of ensuring that every behavior it does generate satisfies the specification. However, for reasons of efficiency, TLC doesn't always meet this goal. It deviates from the semantics of TLA<sup>+</sup> in two ways.

The first deviation is that TLC doesn't preserve the precise semantics of CHOOSE. As explained in Section 16.1, if  $S$  equals  $T$ , then  $\text{CHOOSE } x \in S : P$  should equal  $\text{CHOOSE } x \in T : P$ . However, TLC guarantees this only if  $S$  and  $T$  are syntactically the same. For example, TLC might compute different values for the two expressions

$$\text{CHOOSE } x \in \{1, 2, 3\} : x < 3 \quad \text{CHOOSE } x \in \{3, 2, 1\} : x < 3$$

A similar violation of the semantics of TLA<sup>+</sup> exists with CASE expressions, whose semantics are defined (in Section 16.1.4) in terms of CHOOSE.

The second part of the semantics of TLA<sup>+</sup> that TLC does not preserve is the representation of strings. In TLA<sup>+</sup>, the string “abc” is a three-element sequence—that is, a function with domain  $\{1, 2, 3\}$ . TLC treats strings as primitive values, not as functions. It thus considers the legal TLA<sup>+</sup> expression “abc”[2] to be an error.

## 14.7 The Fine Print

This section describes in detail two aspects of TLC that were sketched above: the grammar of the configuration file, and the precise definition of TLC values.

### 14.7.1 The Grammar of the Configuration File

The grammar of TLC's configuration file is described in the TLA<sup>+</sup> module *ConfigFileGrammar* in Figure 14.6 on the next page. More precisely, the set of sentences *ConfigGrammar.File*, where *ConfigGrammar* is defined in the module, describes all syntactically correct configuration files from which comments have been removed. The *ConfigFileGrammar* module extends the *BNFGrammars* module, which is explained above in Section 11.1.4 (page 179).

Here are some additional restrictions on the configuration file that are not specified by module *ConfigFileGrammar*. There can be at most one INIT and

---

MODULE *ConfigFileGrammar*

---

EXTENDS *BNFGrammars*

---

LEXEMES

---

*Letter*  $\triangleq$  *OneOf*(“abcdefghijklmnopqrstuvwxyz\_ ABCDEFGHIJKLMNOPQRSTUVWXYZ”)

*Num*  $\triangleq$  *OneOf*(“0123456789”)

*LetterOrNum*  $\triangleq$  *Letter*  $\cup$  *Num*

*AnyChar*  $\triangleq$  *LetterOrNum*  $\cup$  *OneOf*(“~ ! @ # \\$ % ^ & \* - + = | ( ) { } [ ] ; : ; ' < > . ? / ”)

*SingularKW*  $\triangleq$  {“SPECIFICATION”, “INIT”, “NEXT”, “VIEW”, “SYMMETRY”}

*PluralKW*  $\triangleq$  {“CONSTRAINT”, “CONSTRAINTS”, “ACTION-CONSTRAINT”, “ACTION-CONSTRAINTS”, “INVARIANT”, “INVARIANTS”, “PROPERTY”, “PROPERTIES”}

*Keyword*  $\triangleq$  *SingularKW*  $\cup$  *PluralKW*  $\cup$  {“CONSTANT”, “CONSTANTS”}

*AnyIdent*  $\triangleq$  *LetterOrNum*<sup>\*</sup> & *Letter* & *LetterOrNum*<sup>\*</sup>

*Ident*  $\triangleq$  *AnyIdent* \ *Keyword*

---

*ConfigGrammar*  $\triangleq$  THE BNF GRAMMAR

---

LET *P*(*G*)  $\triangleq$

$\wedge$  *G.File* ::= *G.Statement*<sup>+</sup>

$\wedge$  *G.Statement* ::= *Tok*(*SingularKW*) & *Tok*(*Ident*)

$|$  *Tok*(*PluralKW*) & *Tok*(*Ident*)<sup>\*</sup>

$|$  *Tok*({“CONSTANT”, “CONSTANTS”})

& (*G.Replacement* | *G.Assignment*)<sup>\*</sup>

$\wedge$  *G.Replacement* ::= *Tok*(*Ident*) & *tok*(“<−”) & *Tok*(*AnyIdent*)

$\wedge$  *G.Assignment* ::= *Tok*(*Ident*) & *tok*(“=”) & *G.IdentValue*

$\wedge$  *G.IdentValue* ::= *Tok*(*AnyIdent*) | *G.Number* | *G.String*

$|$  *tok*(“{”)

& (*Nil* | *G.IdentValue* & (*tok*(“,”) & *G.IdentValue*)<sup>\*</sup>)

& *tok*(“}”)

$\wedge$  *G.Number* ::= (*Nil* | *tok*(“−”)) & *Tok*(*Num*<sup>+</sup>)

$\wedge$  *G.String* ::= *tok*(“ ”) & *Tok*(*AnyChar*<sup>\*</sup>) & *tok*(“ ”)

---

IN *LeastGrammar*(*P*)

---

Figure 14.6: The BNF grammar of the configuration file.

one NEXT statement. There can be one SPECIFICATION statement, but only if there is no INIT or NEXT statement. (See page 243 in Section 14.3.1 for conditions on when these statements must appear.) There can be at most one VIEW statement and at most one SYMMETRY statement. Multiple instances of other statements are allowed. For example, the two statements

```
INVARIANT Inv1
INVARIANT Inv2 Inv3
```

specify that TLC is to check the three invariants *Inv1*, *Inv2*, and *Inv3*. These statements are equivalent to the single statement

```
INVARIANT Inv1 Inv2 Inv3
```

### 14.7.2 Comparable TLC Values

Section 14.2.1 (page 230) describes TLC values. That description is incomplete because it does not define exactly when values are comparable. The precise definition is that two TLC values are comparable iff the following rules imply that they are:

1. Two primitive values are comparable iff they have the same value type.  
This rule implies that “abc” and “123” are comparable, but “abc” and 123 are not.
2. A model value is comparable with any value. (It is equal only to itself.)
3. Two sets are comparable if they have different numbers of elements, or if they have the same numbers of elements and all the elements in one set are comparable with all the elements in the other.  
This rule implies that {1} and {“a”, “b”} are comparable and that {1, 2} and {2, 3} are comparable. However, {1, 2} and {“a”, “b”} are not comparable.
4. Two functions  $f$  and  $g$  are comparable iff (i) their domains are comparable and (ii) if their domains are equal, then  $f[x]$  and  $g[x]$  are comparable for every element  $x$  in their domain.

This rule implies that  $\langle 1, 2 \rangle$  and  $\langle “a”, “b”, “c” \rangle$  are comparable, and that  $\langle 1, “a” \rangle$  and  $\langle 2, “bc” \rangle$  are comparable. However,  $\langle 1, 2 \rangle$  and  $\langle “a”, “b” \rangle$  are not comparable.

# Part IV

# The TLA<sup>+</sup> Language



---

This part of the book describes TLA<sup>+</sup> in detail. Chapter 15 explains the syntax; Chapters 16 and 17 explain the semantics; and Chapter 18 contains the standard modules. Almost all of the TLA<sup>+</sup> language has already been described—mainly through examples. In fact, most of the language was described in Chapters 1–6. This part gives complete specification of the language.

A completely formal specification of TLA<sup>+</sup> would consist of a formal definition of the set of legal (syntactically well-formed) modules, and a precisely defined meaning operator that assigns to every legal module  $M$  its mathematical meaning  $\llbracket M \rrbracket$ . Such a specification would be quite long and of limited interest. Instead, I have tried to provide a fairly informal specification that is detailed enough to show mathematically sophisticated readers how they could write a completely formal one.

These chapters are heavy going, and few people will want to read them completely. However, I hope they can serve as a reference manual for anyone who reads or writes TLA<sup>+</sup> specifications. If you have a question about the finer details of the syntax or the meaning of some part of the language, you should be able to find the answer here.

Tables 1–8 on the next page through page 273 provide a tiny reference manual. Tables 1–4 very briefly describe all the built-in operators of TLA<sup>+</sup>. Table 5 lists all the user-definable operator symbols and indicates which ones are already used by the standard modules. It's a good place to look when choosing notation for your specification. Table 6 gives the precedence of the operators; it is explained in Section 15.2.1 on page 283. Table 7 lists all operators defined by the standard modules. Finally, Table 8 shows how to type any symbol that doesn't have an obvious ASCII equivalent.

## Logic

$\wedge \vee \neg \Rightarrow \equiv$

TRUE FALSE BOOLEAN [the set {TRUE, FALSE}]

$\forall x : p \quad \exists x : p \quad \forall x \in S : p \quad \exists x \in S : p$  <sup>(1)</sup>

CHOOSE  $x : p$  [An  $x$  satisfying  $p$ ] CHOOSE  $x \in S : p$  [An  $x \in S$  satisfying  $p$ ]

## Sets

$= \neq \in \notin \cup \cap \subseteq \setminus$  [set difference]

$\{e_1, \dots, e_n\}$  [Set consisting of elements  $e_i$ ]

$\{x \in S : p\}$  <sup>(2)</sup> [Set of elements  $x \in S$  satisfying  $p$ ]

$\{e : x \in S\}$  <sup>(1)</sup> [Set of elements  $e$  such that  $x \in S$ ]

SUBSET  $S$  [Set of subsets of  $S$ ]

UNION  $S$  [Union of all elements of  $S$ ]

## Functions

$f[e]$  [Function application]

DOMAIN  $f$  [Domain of function  $f$ ]

$[x \in S \mapsto e]$  <sup>(1)</sup> [Function  $f$  such that  $f[x] = e$  for  $x \in S$ ]

$[S \rightarrow T]$  [Set of functions  $f$  with  $f[x] \in T$  for  $x \in S$ ]

$[f \text{ EXCEPT } !e_1 = e_2]$  <sup>(3)</sup> [Function  $\hat{f}$  equal to  $f$  except  $\hat{f}[e_1] = e_2$ ]

## Records

$e.h$  [The  $h$ -field of record  $e$ ]

$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$  [The record whose  $h_i$  field is  $e_i$ ]

$[h_1 : S_1, \dots, h_n : S_n]$  [Set of all records with  $h_i$  field in  $S_i$ ]

$[r \text{ EXCEPT } .h = e]$  <sup>(3)</sup> [Record  $\hat{r}$  equal to  $r$  except  $\hat{r}.h = e$ ]

## Tuples

$e[i]$  [The  $i^{\text{th}}$  component of tuple  $e$ ]

$\langle e_1, \dots, e_n \rangle$  [The  $n$ -tuple whose  $i^{\text{th}}$  component is  $e_i$ ]

$S_1 \times \dots \times S_n$  [The set of all  $n$ -tuples with  $i^{\text{th}}$  component in  $S_i$ ]

## Strings and Numbers

$\text{``c}_1 \dots \text{c}_n\text{''}$  [A literal string of  $n$  characters]

STRING [The set of all strings]

$d_1 \dots d_n \quad d_1 \dots d_n . d_{n+1} \dots d_m$  [Numbers (where the  $d_i$  are digits)]

(1)  $x \in S$  may be replaced by a comma-separated list of items  $v \in S$ , where  $v$  is either a comma-separated list or a tuple of identifiers.

(2)  $x$  may be an identifier or tuple of identifiers.

(3)  $![e_1]$  or  $!.h$  may be replaced by a comma separated list of items  $!a_1 \dots a_n$ , where each  $a_i$  is  $[e_i]$  or  $.h_i$ .

**Table 1:** The constant operators.

|                                                                                                      |                                                                                                       |
|------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| IF $p$ THEN $e_1$ ELSE $e_2$                                                                         | $[e_1 \text{ if } p \text{ true, else } e_2]$                                                         |
| CASE $p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n$                                   | $[\text{Some } e_i \text{ such that } p_i \text{ true}]$                                              |
| CASE $p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n \sqcap \text{OTHER} \rightarrow e$ | $[\text{Some } e_i \text{ such that } p_i \text{ true, or } e \text{ if all } p_i \text{ are false}]$ |
| LET $d_1 \stackrel{\Delta}{=} e_1 \dots d_n \stackrel{\Delta}{=} e_n$ IN $e$                         | $[e \text{ in the context of the definitions}]$                                                       |
| $\wedge p_1$ [the conjunction $p_1 \wedge \dots \wedge p_n$ ]                                        | $\vee p_1$ [the disjunction $p_1 \vee \dots \vee p_n$ ]                                               |
| $\vdots$                                                                                             | $\vdots$                                                                                              |
| $\wedge p_n$                                                                                         | $\vee p_n$                                                                                            |

**Table 2:** Miscellaneous constructs.

|                       |                                                 |
|-----------------------|-------------------------------------------------|
| $e'$                  | [The value of $e$ in the final state of a step] |
| $[A]_e$               | $[A \vee (e' = e)]$                             |
| $\langle A \rangle_e$ | $[A \wedge (e' \neq e)]$                        |
| ENABLED $A$           | [An $A$ step is possible]                       |
| UNCHANGED $e$         | $[e' = e]$                                      |
| $A \cdot B$           | [Composition of actions]                        |

**Table 3:** Action operators.

|                                 |                                                         |
|---------------------------------|---------------------------------------------------------|
| $\square F$                     | $[F \text{ is always true}]$                            |
| $\diamond F$                    | $[F \text{ is eventually true}]$                        |
| $\text{WF}_e(A)$                | $[\text{Weak fairness for action } A]$                  |
| $\text{SF}_e(A)$                | $[\text{Strong fairness for action } A]$                |
| $F \rightsquigarrow G$          | $[F \text{ leads to } G]$                               |
| $F \stackrel{+}{\Rightarrow} G$ | $[F \text{ guarantees } G]$                             |
| $\exists x : F$                 | $[\text{Temporal existential quantification (hiding)}]$ |
| $\forall x : F$                 | $[\text{Temporal universal quantification}]$            |

**Table 4:** Temporal operators.

## Infix Operators

|               |       |               |       |               |         |               |          |           |       |                |
|---------------|-------|---------------|-------|---------------|---------|---------------|----------|-----------|-------|----------------|
| $+$           | $(1)$ | $-$           | $(1)$ | $*$           | $(1)$   | $/$           | $(2)$    | $\circ$   | $(3)$ | $++$           |
| $\div$        | $(1)$ | $\%$          | $(1)$ | $\wedge$      | $(1,4)$ | $\dots$       | $(1)$    | $\dots$   |       | $--$           |
| $\oplus$      | $(5)$ | $\ominus$     | $(5)$ | $\otimes$     |         | $\oslash$     |          | $\odot$   |       | $**$           |
| $<$           | $(1)$ | $>$           | $(1)$ | $\leq$        | $(1)$   | $\geq$        | $(1)$    | $\sqcap$  |       | $//$           |
| $\curlywedge$ |       | $\curlywedge$ |       | $\curlywedge$ |         | $\curlywedge$ |          | $\sqcup$  |       | $\wedge\wedge$ |
| $\ll$         |       | $\gg$         |       | $<:$          |         | $:>$          | $(6)$    | $\&$      |       | $\&\&$         |
| $\sqcap$      |       | $\sqcup$      |       | $\sqcap$      | $(5)$   | $\sqcap$      |          | $ $       |       | $  $           |
| $\sqsubset$   |       | $\sqsupset$   |       |               |         | $\sqcap$      | $\sqcap$ | $\star$   |       | $\%%$          |
| $\vdash$      |       | $\dashv$      |       | $\mp$         |         | $\mp$         | $\mp$    | $\bullet$ |       | $\#\#$         |
| $\sim$        |       | $\approx$     |       | $\approx$     |         | $\approx$     |          | $\$$      |       | $\$\$$         |
| $::=$         |       | $::=$         |       | $\approx$     |         | $\equiv$      |          | $??$      |       | $!!$           |
| $\infty$      |       | $\wr$         |       | $\oplus$      |         | $\circ$       |          | $@@$      | $(6)$ |                |

## Postfix Operators <sup>(7)</sup>

$\hat{+}$        $\hat{*}$        $\hat{\#}$

## Prefix Operator

$\_$  <sup>(8)</sup>

(1) Defined by the *Naturals*, *Integers*, and *Reals* modules.

(2) Defined by the *Reals* module.

(3) Defined by the *Sequences* module.

(4)  $x^y$  is printed as  $x^y$ .

(5) Defined by the *Bags* module.

(6) Defined by the *TLC* module.

(7)  $e\hat{+}$  is printed as  $e^+$ , and similarly for  $\hat{*}$  and  $\hat{\#}$ .

(8) Defined by the *Integers* and *Reals* modules.

**Table 5:** User-definable operator symbols.

### Prefix Operators

|           |      |            |      |        |       |
|-----------|------|------------|------|--------|-------|
| $\neg$    | 4-4  | $\square$  | 4-15 | UNION  | 8-8   |
| ENABLED   | 4-15 | $\diamond$ | 4-15 | DOMAIN | 9-9   |
| UNCHANGED | 4-15 | SUBSET     | 8-8  | -      | 12-12 |

### Infix Operators

|                     |         |                        |          |                |           |                |           |
|---------------------|---------|------------------------|----------|----------------|-----------|----------------|-----------|
| $\Rightarrow$       | 1-1     | $\leq$                 | 5-5      | $<:$           | 7-7       | $\ominus$      | 11-11 (a) |
| $\pm\triangleright$ | 2-2     | $\ll$                  | 5-5      | $\backslash$   | 8-8       | -              | 11-11 (a) |
| $\equiv$            | 2-2     | $\prec$                | 5-5      | $\cap$         | 8-8 (a)   | --             | 11-11 (a) |
| $\rightsquigarrow$  | 2-2     | $\succ$                | 5-5      | $\cup$         | 8-8 (a)   | $\&$           | 13-13 (a) |
| $\wedge$            | 3-3 (a) | $\bowtie$              | 5-5      | $\ldots$       | 9-9       | $\&\&$         | 13-13 (a) |
| $\vee$              | 3-3 (a) | $\sim$                 | 5-5      | $\ldots\ldots$ | 9-9       | $\odot$        | 13-13 (a) |
| $\neq$              | 5-5     | $\simeq$               | 5-5      | $!!$           | 9-13      | $\oslash$      | 13-13     |
| $\dashv$            | 5-5     | $\sqcap$               | 5-5      | $\#\#$         | 9-13 (a)  | $\otimes$      | 13-13 (a) |
| $::=$               | 5-5     | $\sqsubseteq$          | 5-5      | $\$$           | 9-13 (a)  | *              | 13-13 (a) |
| $::=$               | 5-5     | $\sqsupseteq$          | 5-5      | $\$\$$         | 9-13 (a)  | **             | 13-13 (a) |
| $<$                 | 5-5     | $\sqsubset$            | 5-5      | $??$           | 9-13 (a)  | /              | 13-13     |
| $=$                 | 5-5     | $\sqsupset$            | 5-5      | $\sqcap$       | 9-13 (a)  | //             | 13-13     |
| $\equiv$            | 5-5     | $\sqsubseteq\sqsupset$ | 5-5      | $\sqcup$       | 9-13 (a)  | $\bigcirc$     | 13-13 (a) |
| $\triangleright$    | 5-5     | $\succ\prec$           | 5-5      | $\oplus$       | 9-13 (a)  | $\bullet$      | 13-13 (a) |
| $\approx$           | 5-5     | $\succ\sim$            | 5-5      | $\wr$          | 9-14      | $\div$         | 13-13     |
| $\asymp$            | 5-5     | $\sim\prec$            | 5-5      | $\oplus$       | 10-10 (a) | $\circ$        | 13-13 (a) |
| $\cong$             | 5-5     | $\sim\sqsupset$        | 5-5      | $+$            | 10-10 (a) | $\star$        | 13-13 (a) |
| $\doteq$            | 5-5     | $\sqsupset\sqsubset$   | 5-5      | $++$           | 10-10 (a) | $\wedge$       | 14-14     |
| $\geq$              | 5-5     | $\sqsupseteq$          | 5-5      | $\%$           | 10-11     | $\wedge\wedge$ | 14-14     |
| $\gg$               | 5-5     | $\cdot^{(1)}$          | 5-14 (a) | $\%\%$         | 10-11 (a) | $\cdot^{(2)}$  | 17-17 (a) |
| $\in$               | 5-5     | $\text{@}@$            | 6-6 (a)  |                | 10-11 (a) |                |           |
| $\notin$            | 5-5     | $:$                    | > 7-7    | $\parallel$    | 10-11 (a) |                |           |

### Postfix Operators

$\wedge\wedge$  15-15     $\wedge\wedge\wedge$  15-15     $\wedge\wedge\wedge\wedge$  15-15    ' 15-15

(1) Action composition ( $\cdot$ ).

(2) Record field (period).

**Table 6:** The precedence ranges of operators. The relative precedence of two operators is unspecified if their ranges overlap. Left-associative operators are indicated by (a).

**Modules** *Naturals*, *Integers*, *Reals*

|        |           |        |           |                |      |                           |                                |
|--------|-----------|--------|-----------|----------------|------|---------------------------|--------------------------------|
| $+$    | $-^{(1)}$ | $*$    | $/^{(2)}$ | $\wedge^{(3)}$ | $..$ | <i>Nat</i>                | <i>Real</i> <sup>(2)</sup>     |
| $\div$ | $\%$      | $\leq$ | $\geq$    | $<$            | $>$  | <i>Int</i> <sup>(4)</sup> | <i>Infinity</i> <sup>(2)</sup> |

(1) Only infix  $-$  is defined in *Naturals*.(2) Defined only in *Reals* module.

(3) Exponentiation.

(4) Not defined in *Naturals* module.**Module** *Sequences*

|         |               |             |                  |               |
|---------|---------------|-------------|------------------|---------------|
| $\circ$ | <i>Append</i> | <i>Head</i> | <i>SelectSeq</i> | <i>SubSeq</i> |
|         |               | <i>Len</i>  | <i>Seq</i>       | <i>Tail</i>   |

**Module** *FiniteSets*

|                    |                    |
|--------------------|--------------------|
| <i>IsFiniteSet</i> | <i>Cardinality</i> |
|--------------------|--------------------|

**Module** *Bags*

|                       |                 |                 |               |
|-----------------------|-----------------|-----------------|---------------|
| $\oplus$              | <i>BagIn</i>    | <i>CopiesIn</i> | <i>SubBag</i> |
| $\ominus$             | <i>BagOfAll</i> | <i>EmptyBag</i> |               |
| $\sqsubseteq$         | <i>BagToSet</i> | <i>IsABag</i>   |               |
| <i>BagCardinality</i> | <i>BagUnion</i> | <i>SetToBag</i> |               |

**Module** *RealTime*

|                |              |                                        |
|----------------|--------------|----------------------------------------|
| <i>RTBound</i> | <i>RTnow</i> | <i>now</i> (declared to be a variable) |
|----------------|--------------|----------------------------------------|

**Module** *TLC*

|      |      |                |               |                 |                     |
|------|------|----------------|---------------|-----------------|---------------------|
| $:>$ | $@@$ | <i>Print</i>   | <i>Assert</i> | <i>JavaTime</i> | <i>Permutations</i> |
|      |      | <i>SortSeq</i> |               |                 |                     |

**Table 7:** Operators defined in the standard modules.

|               |                                                 |               |                                         |                    |                                     |
|---------------|-------------------------------------------------|---------------|-----------------------------------------|--------------------|-------------------------------------|
| $\wedge$      | $\wedge$ or $\backslash land$                   | $\vee$        | $\vee$ or $\backslash lor$              | $\Rightarrow$      | $\Rightarrow$                       |
| $\neg$        | $\sim$ or $\backslash lnot$ or $\backslash neg$ | $\equiv$      | $\Leftrightarrow$ or $\backslash equiv$ | $\triangleq$       | $\equiv$                            |
| $\in$         | $\backslash in$                                 | $\notin$      | $\backslash notin$                      | $\neq$             | $\#$ or $/=$                        |
| $\langle$     | $\langle\langle$                                | $\rangle$     | $\gg$                                   | $\square$          | $[]$                                |
| $<$           | $<$                                             | $\rangle$     | $>$                                     | $\diamond$         | $\Leftrightarrow$                   |
| $\leq$        | $\backslash leq$ or $=<$ or $\leq$              | $\geq$        | $\backslash geq$ or $\geq$              | $\rightsquigarrow$ | $\sim\!>$                           |
| $\ll$         | $\backslash ll$                                 | $\gg$         | $\backslash gg$                         | $\pm\Rightarrow$   | $\dashleftarrow$                    |
| $\prec$       | $\backslash prec$                               | $\succ$       | $\backslash succ$                       | $\mapsto$          | $ -\>$                              |
| $\preceq$     | $\backslash preceq$                             | $\succeq$     | $\backslash succceq$                    | $\div$             | $\backslash div$                    |
| $\subseteq$   | $\backslash subseq$                             | $\supseteq$   | $\backslash supseteq$                   | $\cdot$            | $\backslash cdot$                   |
| $\subset$     | $\backslash subset$                             | $\supset$     | $\backslash supset$                     | $\circ$            | $\backslash o$ or $\backslash circ$ |
| $\sqsubset$   | $\backslash sqsubset$                           | $\sqsupset$   | $\backslash sqsupset$                   | $\bullet$          | $\backslash bullet$                 |
| $\sqsubseteq$ | $\backslash sqsubseteq$                         | $\sqsupseteq$ | $\backslash sqsupseteq$                 | $\star$            | $\backslash star$                   |
| $\vdash$      | $\vdash$                                        | $\dashv$      | $- $                                    | $\bigcirc$         | $\backslash bigcirc$                |
| $\models$     | $\models$                                       | $\models$     | $= $                                    | $\sim$             | $\backslash sim$                    |
| $\rightarrow$ | $\rightarrow$                                   | $\leftarrow$  | $<-$                                    | $\approx$          | $\backslash simeq$                  |
| $\cap$        | $\backslash cap$ or $\backslash intersect$      | $\cup$        | $\backslash cup$ or $\backslash union$  | $\asymp$           | $\backslash asymp$                  |
| $\sqcap$      | $\backslash sqcap$                              | $\sqcup$      | $\backslash sqcup$                      | $\approx$          | $\backslash approx$                 |
| $\oplus$      | $(+)$ or $\backslash oplus$                     | $\sqcup$      | $\backslash uplus$                      | $\cong$            | $\backslash cong$                   |
| $\ominus$     | $(-)$ or $\backslash ominus$                    | $\times$      | $\backslash X$ or $\backslash times$    | $\doteq$           | $\backslash doteq$                  |
| $\odot$       | $(.)$ or $\backslash odot$                      | $\wr$         | $x^y$                                   | $x^y$              | $x^y$ <sup>(2)</sup>                |
| $\otimes$     | $(\backslash X)$ or $\backslash otimes$         | $\propto$     | $x^+$                                   | $x^+$              | $x^+$ <sup>(2)</sup>                |
| $\oslash$     | $(/)$ or $\backslash oslash$                    | $s$           | $x^s$                                   | $x^*$              | $x^*$ <sup>(2)</sup>                |
| $\exists$     | $\backslash E$                                  | $\forall$     | $\backslash A$                          | $x^\#$             | $x^\#$ <sup>(2)</sup>               |
| $\exists$     | $\backslash EE$                                 | $\forall$     | $\backslash AA$                         | $,$                | $,$                                 |
| $]_v$         | $]_v$                                           | $\rangle_v$   | $\gg_v$                                 |                    |                                     |
| $WF_v$        | $WF_v$                                          | $SF_v$        | $SF_v$                                  |                    |                                     |
|               | ----- (3)                                       |               | ----- (3)                               |                    |                                     |
|               | ----- (3)                                       |               | ===== (3)                               |                    |                                     |

(1)  $s$  is a sequence of characters. See Section 16.1.10 on page 307.

(2)  $x$  and  $y$  are any expressions.

(3) a sequence of four or more  $-$  or  $=$  characters.

**Table 8:** The ASCII representations of typeset symbols.



# Chapter 15

## The Syntax of TLA<sup>+</sup>

This book uses the ASCII version of TLA<sup>+</sup>—the version based on the ASCII character set. One can define other versions of TLA<sup>+</sup> that use different sets of characters. A different version might allow an expression like  $\Omega[\ddot{a}] \circ \langle \text{“ca”} \rangle$ . Since mathematical formulas look pretty much the same in most languages, the basic syntax of all versions of TLA<sup>+</sup> should be the same. Different versions would differ in their lexemes and in the identifiers and strings they allow. This chapter describes the syntax of the ASCII version, the only one that now exists.

The term *syntax* has two different usages, which I will somewhat arbitrarily attribute to mathematicians and computer scientists. A computer scientist would say that  $\langle a, a \rangle$  is a syntactically correct TLA<sup>+</sup> expression. A mathematician would say that the expression is syntactically correct iff it appears in a context in which  $a$  is defined or declared. A computer scientist would call this requirement a *semantic* rather than a syntactic condition. A mathematician would say that  $\langle a, a \rangle$  is meaningless if  $a$  isn’t defined or declared, and one can’t talk about the semantics of a meaningless expression. This chapter describes the syntax of TLA<sup>+</sup>, in the computer scientist’s sense of syntax. The “semantic” part of the syntax is specified in Chapters 16 and 17.

TLA<sup>+</sup> is designed to be easy for humans to read and write. In particular, its syntax for expressions tries to capture some of the richness of ordinary mathematical notation. This makes a precise specification of the syntax rather complicated. Such a specification has been written in TLA<sup>+</sup>, but it’s quite detailed and you probably don’t want to look at it unless you are writing a parser for the language. This chapter gives a less formal description of the syntax that should answer any questions likely to arise in practice. Section 15.1 specifies precisely a simple grammar that ignores some aspects of the syntax such as operator precedence, indentation rules for  $\wedge$  and  $\vee$  lists, and comments. These other aspects are explained informally in Section 15.2. Sections 15.1 and 15.2 describe the grammar of a TLA<sup>+</sup> module viewed as a sequence of lexemes, where

a lexeme is a sequence of characters such as `|→` that forms an atomic unit of the grammar. Section 15.3 describes how the sequence of characters that you actually type are turned into a sequence of lexemes. It includes the precise syntax for comments.

This chapter describes the ASCII syntax for TLA<sup>+</sup> specifications. Typeset versions of specifications appear in this book. For example, the infix operator typeset as `prec` is represented in ASCII as `\prec`. Table 8 on page 273 gives the correspondence between the ASCII and typeset versions of all TLA<sup>+</sup> symbols for which the correspondence may not be obvious.

## 15.1 The Simple Grammar

The simple grammar of TLA<sup>+</sup> is described in BNF. More precisely, it is specified below in the TLA<sup>+</sup> module *TLAPlusGrammar*. This module uses the operators for representing BNF grammars defined in the *BNFGrammars* module of Section 11.1.4 (page 179). Module *TLAPlusGrammar* contains comments describing how to read the specification as an ordinary BNF grammar. So, if you are familiar with BNF grammars and just want to learn the syntax of TLA<sup>+</sup>, you don't have to understand how the TLA<sup>+</sup> operators for writing grammars are defined. Otherwise, you should read Section 11.1.4 before trying to read the following module.

---

MODULE *TLAPlusGrammar*

---

EXTENDS *Naturals*, *Sequences*, *BNFGrammars*

---

This module defines a simple grammar for TLA<sup>+</sup> that ignores many aspects of the language, such as operator precedence and indentation rules. I use the term *sentence* to mean a sequence of lexemes, where a lexeme is just a string. The *BNFGrammars* module defines the following standard conventions for writing sets of sentences:  $L \mid M$  means an  $L$  or an  $M$ ,  $L^*$  means the concatenation of zero or more  $L$ s, and  $L^+$  means the concatenation of one or more  $L$ s. The concatenation of an  $L$  and an  $M$  is denoted by  $L \& M$  rather than the customary juxtaposition  $LM$ . *Nil* is the null sentence, so *Nil*  $\&$   $L$  equals  $L$  for any  $L$ .

A *token* is a one-lexeme sentence. There are two operators for defining sets of tokens: if  $s$  is a lexeme, then  $\text{tok}(s)$  is the set containing the single token  $\langle s \rangle$ ; and if  $S$  is a set of lexemes, then  $\text{Tok}(S)$  is the set containing all tokens  $\langle s \rangle$  for  $s \in S$ . In comments, I will not distinguish between the token  $\langle s \rangle$  and the string  $s$ .

We begin by defining two useful operators. First, a *CommaList*( $L$ ) is defined to be an  $L$  or a sequence of  $L$ s separated by commas.

$$\text{CommaList}(L) \triangleq L \& (\text{tok}(,)) \& L^*$$

Next, if  $c$  is a character, then we define *AtLeast4*(“ $c$ ”) to be the set of tokens consisting of 4 or more  $c$ 's.

$$\text{AtLeast4}(s) \triangleq \text{Tok}(\{s \circ s \circ s\} \& \{s\}^+)$$


---

We now define some sets of lexemes. First is *ReservedWord*, the set of words that can't be used as identifiers. (Note that BOOLEAN, TRUE, FALSE, and STRING are identifiers that are predefined.)

$$\begin{aligned} \text{ReservedWord} &\triangleq \\ \{ & \text{“ASSUME”, “ELSE”, “LOCAL”, “UNION”,} \\ & \text{“ASSUMPTION”, “ENABLED”, “MODULE”, “VARIABLE”,} \\ & \text{“AXIOM”, “EXCEPT”, “OTHER”, “VARIABLES”,} \\ & \text{“CASE”, “EXTENDS”, “SF_”, “WF_”,} \\ & \text{“CHOOSE”, “IF”, “SUBSET”, “WITH”,} \\ & \text{“CONSTANT”, “IN”, “THEN”,} \\ & \text{“CONSTANTS”, “INSTANCE”, “THEOREM”,} \\ & \text{“DOMAIN”, “LET”, “UNCHANGED”} \} \end{aligned}$$

Next are three sets of characters—more precisely, sets of 1-character lexemes. They are the sets of letters, numbers, and characters that can appear in an identifier.

$$\begin{aligned} \text{Letter} &\triangleq \\ \text{OneOf}(& \text{“abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ”}) \\ \text{Numeral} &\triangleq \text{OneOf}(\text{“0123456789”}) \\ \text{NameChar} &\triangleq \text{Letter} \cup \text{Numeral} \cup \{“_”\} \end{aligned}$$

We now define some sets of tokens. A *Name* is a token composed of letters, numbers, and `_` characters that contains at least one letter, but does not begin with `WF_` or `SF_` (see page 290 for an explanation of this restriction). It can be used as the name of a record field or a module. An *Identifier* is a *Name* that isn't a reserved word.

$$\begin{aligned} \text{Name} &\triangleq \text{Tok}((\text{NameChar}^* \& \text{Letter} \& \text{NameChar}^*) \\ &\quad \backslash (\{\text{“WF_”, “SF_”}\} \& \text{NameChar}^+)) \\ \text{Identifier} &\triangleq \text{Name} \backslash \text{Tok}(\text{ReservedWord}) \end{aligned}$$

An *IdentifierOrTuple* is either an identifier or a tuple of identifiers. Note that `( )` is typed as `<< >>`.

$$\begin{aligned} \text{IdentifierOrTuple} &\triangleq \\ \text{Identifier} &\mid \text{tok}(\text{“<<”}) \& \text{CommaList}(\text{Identifier}) \& \text{tok}(\text{“>>”}) \end{aligned}$$

A *Number* is a token representing a number. You can write the integer 63 in the following ways: 63, 63.00, `\b111111` or `\B111111` (binary), `\o77` or `\O77` (octal), or `\h3f`, `\H3f`, `\h3F`, or `\H3F` (hexadecimal).

$$\begin{aligned} \text{NumberLexeme} &\triangleq \text{Numeral}^+ \\ &\mid (\text{Numeral}^* \& \{“.”\} \& \text{Numeral}^+) \\ &\mid \{“\b”, “\B”\} \& \text{OneOf}(\text{“01”})^+ \\ &\mid \{“\o”, “\O”\} \& \text{OneOf}(\text{“01234567”})^+ \\ &\mid \{“\h”, “\H”\} \& \text{OneOf}(\text{“0123456789abcdefABCDEF”})^+ \end{aligned}$$

$$\text{Number} \triangleq \text{Tok}(\text{NumberLexeme})$$

A *String* token represents a literal string. See Section 16.1.10 on page 307 to find out how special characters are typed in a string.

$$String \triangleq Tok(\{"\text{ "}\} \& STRING \& \{"\text{ "}\})$$

We next define the sets of tokens that represent prefix operators (like  $\square$ ), infix operators (like  $+$ ), and postfix operators (like prime ('')). See Table 8 on page 273 to find out what symbols these ASCII strings represent.

$$\textit{PrefixOp} \triangleq \textit{Tok}(\{ \text{``-''}, \text{``~''}, \text{``\textbackslash lnot''}, \text{``\textbackslash neg''}, \text{``[]''}, \text{``<>''}, \text{``DOMAIN''}, \text{``ENABLED''}, \text{``SUBSET''}, \text{``UNCHANGED''}, \text{``UNION''} \})$$

*InfixOp*  $\triangleq$

$$PostfixOp \triangleq Tok(\{ "+", "*", "#", , \})$$

Formally, the grammar *TLAPlusGrammar* of  $\text{TLA}^+$  is the smallest grammar satisfying the BNF productions below.

## TLAPlusGrammar $\triangleq$

LET  $P(G) \triangleq$

Here is the BNF grammar. Terms that begin with “*G*.”, like *G.Module*, represent nonterminals. The terminals are sets of tokens, either defined above or described with the operators *tok* and *Tok*. The operators *AtLeast4* and *CommaList* are defined above.

$\wedge G.Module ::= AtLeast4("-") \& tok("MODULE") \& Name \& AtLeast4("-")$   
 $\quad \& (Nil \mid (tok("EXTENDS") \& CommaList(Name)))$   
 $\quad \& (G.Unit)^*$   
 $\quad \& AtLeast4("=")$

$\wedge G.Unit ::=$   $G.VariableDeclaration$   
 $\quad | G.ConstantDeclaration$   
 $\quad | (Nil \mid tok("LOCAL")) \ \& \ G.OperatorDefinition$   
 $\quad | (Nil \mid tok("LOCAL")) \ \& \ G.FunctionDefinition$   
 $\quad | (Nil \mid tok("LOCAL")) \ \& \ G.Instance$   
 $\quad | (Nil \mid tok("LOCAL")) \ \& \ G.ModuleDefinition$   
 $\quad | G.Assumption$   
 $\quad | G.Theorem$   
 $\quad | G.Module$   
 $\quad | AtLeast4("-")$

$\wedge G.VariableDeclaration ::=$   
 $\quad Tok(\{"VARIABLE", "VARIABLES"\}) \ \& \ CommaList(Identifier)$

$\wedge G.ConstantDeclaration ::=$   
 $\quad Tok(\{"CONSTANT", "CONSTANTS"\}) \ \& \ CommaList(G.OpDecl)$

$\wedge G.OpDecl ::=$   $Identifier$   
 $\quad | Identifier \ \& \ tok("(") \ \& \ CommaList(tok("-")) \ \& \ tok(")")$   
 $\quad | PrefixOp \ \& \ tok("-")$   
 $\quad | tok("-") \ \& \ InfixOp \ \& \ tok("-")$   
 $\quad | tok("-") \ \& \ PostfixOp$

$\wedge G.OperatorDefinition ::=$   $($   $G.NonFixLHS$   
 $\quad | PrefixOp \ \& \ Identifier$   
 $\quad | Identifier \ \& \ InfixOp \ \& \ Identifier$   
 $\quad | Identifier \ \& \ PostfixOp$   $)$   
 $\quad \& \ tok("==")$   
 $\quad \& \ G.Expression$

$\wedge G.NonFixLHS ::=$   
 $\quad Identifier$   
 $\quad \& \ ($   $Nil$   
 $\quad \quad | tok("(") \ \& \ CommaList(Identifier \mid G.OpDecl) \ \& \ tok(")")$   $)$

$\wedge G.FunctionDefinition ::=$   
 $\quad Identifier$   
 $\quad \& \ tok("[") \ \& \ CommaList(G.QuantifierBound) \ \& \ tok("])")$   
 $\quad \& \ tok("==")$   
 $\quad \& \ G.Expression$

$\wedge G.QuantifierBound ::= ( IdentifierOrTuple \mid CommaList(Identifier) )$   
 $\quad \& tok("in")$   
 $\quad \& G.Expression$

$\wedge G.Instance ::= tok("INSTANCE")$   
 $\quad \& Name$   
 $\quad \& ( Nil \mid tok("WITH") \& CommaList(G.Substitution) )$

$\wedge G.Substitution ::= ( Identifier \mid PrefixOp \mid InfixOp \mid PostfixOp )$   
 $\quad \& tok("<-")$   
 $\quad \& G.Argument$

$\wedge G.Argument ::= G.Expression$   
 $\quad \mid G.GeneralPrefixOp$   
 $\quad \mid G.GeneralInfixOp$   
 $\quad \mid G.GeneralPostfixOp$

$\wedge G.InstancePrefix ::=$   
 $\quad ( Identifier$   
 $\quad \& ( Nil$   
 $\quad \mid tok("(") \& CommaList(G.Expression) \& tok(")") )$   
 $\quad \& tok("!) )^*$

$\wedge G.GeneralIdentifier ::= G.InstancePrefix \& Identifier$

$\wedge G.GeneralPrefixOp ::= G.InstancePrefix \& PrefixOp$

$\wedge G.GeneralInfixOp ::= G.InstancePrefix \& InfixOp$

$\wedge G.GeneralPostfixOp ::= G.InstancePrefix \& PostfixOp$

$\wedge G.ModuleDefinition ::= G.NonFixLHS \& tok("==") \& G.Instance$

$\wedge G.Assumption ::= Tok(\{"ASSUME", "ASSUMPTION", "AXIOM"\}) \& G.Expression$

$\wedge G.Theorem ::= tok("THEOREM") \& G.Expression$

The comments give examples of each of the different types of expression.

$\wedge G.Expression ::=$

$G.GeneralIdentifier \quad A(x + 7)!B!Id$

$\mid G.GeneralIdentifier \& tok("(")$   $A!Op(x + 1, y)$   
 $\quad \& CommaList(G.Argument) \& tok(")")$   
 $\mid G.GeneralPrefixOp \& G.Expression \quad \text{SUBSET } S.foo$

- |  $G.Expression \ \& \ G.GeneralInfixOp \ \& \ G.Expression \quad a + b$
- |  $G.Expression \ \& \ G.GeneralPostfixOp \quad x[1]'$
- |  $tok("(") \ \& \ G.Expression \ \& \ tok(")") \quad (x + 1)$
- |  $Tok(\{\backslash A, \backslash E\}) \ \& \ CommaList(G.QuantifierBound) \quad \forall x \in S, \langle y, z \rangle \in T : F(x, y, z)$   
 $\quad \& \ tok(":") \ \& \ G.Expression$
- |  $Tok(\{\backslash A, \backslash E, \backslash AA, \backslash EE\}) \ \& \ CommaList(Identifier) \quad \exists x, y : x + y > 0$   
 $\quad \& \ tok(":") \ \& \ G.Expression$
- |  $tok("CHOOSE") \quad \text{CHOOSE } \langle x, y \rangle \in S : F(x, y)$   
 $\& \ IdentifierOrTuple$   
 $\& \ (Nil \mid tok("\backslash in") \ \& \ G.Expression)$   
 $\& \ tok(":")$   
 $\& \ G.Expression$
- |  $tok("\{") \ \& \ (Nil \mid CommaList(G.Expression)) \ \& \ tok("\}") \quad \{1, 2, 2 + 2\}$
- |  $tok("\{") \quad \{x \in Nat : x > 0\}$   
 $\& \ IdentifierOrTuple \ \& \ tok("\backslash in") \ \& \ G.Expression$   
 $\& \ tok(":")$   
 $\& \ G.Expression$   
 $\& \ tok("\}")$
- |  $tok("\{") \quad \{F(x, y, z) : x, y \in S, z \in T\}$   
 $\& \ G.Expression$   
 $\& \ tok(":")$   
 $\& \ CommaList(G.QuantifierBound)$   
 $\& \ tok("\}")$
- |  $G.Expression \ \& \ tok("[") \ \& \ CommaList(G.Expression) \ \& \ tok("]") \quad f[i + 1, j]$
- |  $tok("[") \quad [i, j \in S, \langle p, q \rangle \in T \mapsto F(i, j, p, q)]$   
 $\& \ CommaList(G.QuantifierBound)$   
 $\& \ tok("\rightarrow")$   
 $\& \ G.Expression$   
 $\& \ tok("]")$
- |  $tok("[") \ \& \ G.Expression \ \& \ tok("\rightarrow") \ \& \ G.Expression \ \& \ tok("]") \quad [(S \cup T) \rightarrow U]$
- |  $tok("[") \ \& \ CommaList( Name \ \& \ tok("\rightarrow") \ \& \ G.Expression ) \quad [a \mapsto x + 1, b \mapsto y]$   
 $\quad \& \ tok("])$

- |  $tok([]) \& CommaList(Name \& tok(:) \& G.Expression) \quad [a : Nat, b : S \cup T]$   
 $\& tok([])$
- |  $tok([]) \quad [f \text{ EXCEPT } ![1, x].r = 4, ![(2, y)] = e]$   
 $\& G.Expression$   
 $\& tok(\text{EXCEPT})$   
 $\& CommaList(tok(!))$   
 $\& (tok(.) \& Name$   
 $\quad | \quad tok([]) \& CommaList(G.Expression) \& tok([]) )^+$   
 $\& tok(=) \& G.Expression)$   
 $\& tok([])$
- |  $tok(<<) \& CommaList(G.Expression) \& tok(>>) \quad \langle 1, 2, 1 + 2 \rangle$
- |  $G.Expression \& (Tok(\{\text{\textbackslash X}, \text{\textbackslash times}\}) \& G.Expression)^+ \quad Nat \times (1 \dots 3) \times Real$
- |  $tok([]) \& G.Expression \& tok([]_+) \& G.Expression \quad [A \vee B]_{(x,y)}$
- |  $tok(<<) \& G.Expression \& tok(>>_+) \& G.Expression \quad \langle x' = y + 1 \rangle_{(x,y)}$
- |  $Tok(\{\text{WF}_-, \text{SF}_-\}) \& G.Expression \quad \text{WF}_{vars}(Next)$   
 $\& tok(()) \& G.Expression \& tok(())$
- |  $tok(\text{IF}) \& G.Expression \& tok(\text{THEN}) \quad \text{IF } p \text{ THEN } A \text{ ELSE } B$   
 $\& G.Expression \& tok(\text{ELSE}) \& G.Expression$
- |  $tok(\text{CASE}) \quad \text{CASE } p1 \rightarrow e1$   
 $\& ( \text{LET } CaseArm \triangleq$   
 $\quad G.Expression \& tok(\rightarrow) \& G.Expression$   
 $\quad \text{IN } CaseArm \& (tok([]) \& CaseArm)^* )$   
 $\& ( \text{Nil}$   
 $\quad | \quad (tok([]) \& tok(\text{OTHER}) \& tok(\rightarrow) \& G.Expression) )$
- |  $tok(\text{LET}) \quad \text{LET } x \triangleq y + 1$   
 $\& ( \quad G.OperatorDefinition$   
 $\quad | \quad G.FunctionDefinition$   
 $\quad | \quad G.ModuleDefinition )^+$   
 $\& tok(\text{IN})$   
 $\& G.Expression$
- |  $(tok(\text{\textbackslash \textbackslash}) \& G.Expression)^+ \quad \wedge x = 1$   
 $\quad \wedge y = 2$
- |  $(tok(\text{\textbackslash /}) \& G.Expression)^+ \quad \vee x = 1$   
 $\quad \vee y = 2$

|  |                 |                                               |
|--|-----------------|-----------------------------------------------|
|  | <i>Number</i>   | 09001                                         |
|  | <i>String</i>   | “foo”                                         |
|  | <i>tok(“@”)</i> | @ (Can be used only in an EXCEPT expression.) |

IN  $\text{LeastGrammar}(P)$

## 15.2 The Complete Grammar

We now complete our explanation of the syntax of  $\text{TLA}^+$  by giving the details that are not described by the BNF grammar in the previous section. Section 15.2.1 gives the precedence rules, Section 15.2.2 gives the alignment rules for conjunction and disjunction lists, and Section 15.2.3 describes comments. Section 15.2.4 briefly discusses the syntax of temporal formulas. Finally, for completeness, Section 15.2.5 explains the handling of two anomalous cases that you’re unlikely ever to encounter.

### 15.2.1 Precedence and Associativity

The expression  $a + b * c$  is interpreted as  $a + (b * c)$  rather than  $(a + b) * c$ . This convention is described by saying that the operator  $*$  has *higher precedence* than the operator  $+$ . In general, operators with higher precedence are applied before operators of lower precedence. This applies to prefix operators (like `SUBSET`) and postfix operators (like `'`) as well as to infix operators like `+` and `*`. Thus,  $a + b'$  is interpreted as  $a + (b')$ , rather than as  $(a + b)'$ , because `'` has higher precedence than `+`. Application order can also be determined by associativity. The expression  $a - b - c$  is interpreted as  $(a - b) - c$  because `-` is a left-associative infix operator.

In  $\text{TLA}^+$ , the precedence of an operator is a range of numbers, like 9–13. The operator `$` has higher precedence than the operator `:>` because the precedence of `$` is 9–13, and this entire range is greater than the precedence range of `:>`, which is 7–7. An expression is illegal (not syntactically well-formed) if the order of application of two operators is not determined because their precedence ranges overlap and they are not two instances of an associative infix operator. For example, the expression  $a + b * c' \% d$  is illegal for the following reason. The precedence range of `'` is higher than that of `*`, and the precedence range of `*` is higher than that of both `+` and `%`, so this expression can be written as  $a + (b * (c')) \% d$ . However, the precedences of `+` (10–10) and `%` (10–11) overlap, so we don’t know if the expression is to be interpreted as  $(a + (b * (c')) \% d$  or  $a + ((b * (c')) \% d)$ , and it is therefore illegal.

TLA<sup>+</sup> embodies the philosophy that it's better to require parentheses than to allow expressions that could easily be misinterpreted. Thus, \* and / have overlapping precedence, making an expression like  $a/b * c$  illegal. (This also makes  $a * b/c$  illegal, even though  $(a * b)/c$  and  $a * (b/c)$  happen to be equal when \* and / have their usual definitions.) Unconventional operators like \$ have wide precedence ranges for safety. But, even when the precedence rules imply that parentheses aren't needed, it's often a good idea to use them anyway if you think there's any chance that a reader might not understand how an expression is parsed.

Table 6 on page 271 gives the precedence ranges of all operators and tells which infix operators are left associative. (No TLA<sup>+</sup> operators are right associative.) Note that the symbols  $\in$ ,  $=$ , and  $.$  are used both as fixed parts of constructs and as infix operators. They are not infix operators in the following two expressions:

$$\{x \in S : p(x)\} \quad [f \text{ EXCEPT } !.a = e]$$

so the precedence of the corresponding infix operators plays no role in parsing these expressions. Below are some additional precedence rules not covered by the operator precedence ranges.

## Function Application

Function application is treated like an operator with precedence range 16–16, giving it higher precedence than any operator except period (".")<sup>1</sup>, the record-field operator. Thus,  $a + b.c[d]'$  is interpreted as  $a + (((b.c)[d])')$ .

## Cartesian Products

In the Cartesian product construct,  $\times$  (typed as \X or \times) acts somewhat like an associative infix operator with precedence range 10–13. Thus,  $A \times B \subseteq C$  is interpreted as  $(A \times B) \subseteq C$ , rather than as  $A \times (B \subseteq C)$ . However,  $\times$  is part of a special construct, not an infix operator. For example, the three sets  $A \times B \times C$ ,  $(A \times B) \times C$ , and  $A \times (B \times C)$  are all different:

$$\begin{aligned} A \times B \times C &= \{\langle a, b, c \rangle : a \in A, b \in B, c \in C\} \\ (A \times B) \times C &= \{\langle \langle a, b \rangle, c \rangle : a \in A, b \in B, c \in C\} \\ A \times (B \times C) &= \{\langle a, \langle b, c \rangle \rangle : a \in A, b \in B, c \in C\} \end{aligned}$$

The first is a set of triples; the last two are sets of pairs.

## Undelimited Constructs

TLA<sup>+</sup> has several expression-making constructs with no explicit right-hand terminator. They are: CHOOSE, IF/THEN/ELSE, CASE, LET/IN, and quantifier constructs. These constructs are treated as prefix operators with the lowest possible

precedence, so an expression made with one of them extends as far as possible. More precisely, the expression is ended only by one of the following:

- The beginning of the next module unit. (Module units are produced by the *Unit* nonterminal in the BNF grammar of Section 11.1.4; they include definition and declaration statements.)
- A right delimiter whose matching left delimiter occurs before the beginning of the construct. Delimiter pairs are ( ), [ ], { }, and ⟨ ⟩.
- Any of the following lexemes, if they are not part of a subexpression: THEN, ELSE, IN, comma (,), colon (:), and →. For example, the subexpression  $\forall x : P$  is ended by the THEN in the expression

IF  $\forall x : P$  THEN 0 ELSE 1

- The CASE separator  $\square$  (not the prefix temporal operator that is typed the same) ends all of these constructs except a CASE statement without an OTHER clause. That is, the  $\square$  acts as a delimiter except when it can be part of a CASE statement.
- Any symbol not to the right of the  $\wedge$  or  $\vee$  prefixing a conjunction or disjunction list element containing the construct. (See Section 15.2.2 on the next page.)

Here is how some expressions are interpreted under this rule:

$$\begin{array}{lll} \text{IF } x > 0 \text{ THEN } y + 1 \\ \quad \quad \quad \text{ELSE } y - 1 & \text{means} & \text{IF } x > 0 \text{ THEN } y + 1 \\ & & \quad \quad \quad \text{ELSE } (y - 1 + 2) \\ + 2 & & \\ \\ \forall x \in S : P(x) & \text{means} & \forall x \in S : (P(x) \vee Q) \\ \vee Q & & \end{array}$$

As these examples show, indentation is ignored—except in conjunction and disjunction lists, discussed below. The absence of a terminating lexeme (an END) for an IF/THEN/ELSE or CASE construct usually makes an expression less cluttered, but sometimes it does require you to add parentheses.

## Subscripts

TLA uses subscript notation in the following constructs:  $[A]_e$ ,  $\langle A \rangle_e$ ,  $\text{WF}_e(A)$ , and  $\text{SF}_e(A)$ . In  $\text{TLA}^+$ , these are written with a “\_” character, as in  $\langle\langle A \rangle\rangle_x$ . This notation is, in principle, problematic. The expression  $\langle\langle A \rangle\rangle_x \wedge B$ , which we expect to mean  $(\langle A \rangle_x) \wedge B$ , could conceivably be interpreted as  $\langle A \rangle_{(x \wedge B)}$ . The precise rule for parsing these constructs isn’t important; you should put parentheses around the subscript except in the following two cases:

- The subscript is a *GeneralIdentifier* in the BNF grammar.
- The subscript is an expression enclosed by one of the following matching delimiter pairs: ( ), [ ], { }, or { }—for example,  $\langle x, y \rangle$  or  $(x + y)$ .

Although  $[A]_f[x]$  is interpreted correctly as  $[A]_{f[x]}$ , it will be easier to read in the ASCII text (and will be formatted properly by TLATE<sub>EX</sub>) if you write it as  $[A]_{\langle f[x] \rangle}$ .

## 15.2.2 Alignment

The most novel aspect of TLA<sup>+</sup> syntax is the aligned conjunction and disjunction lists. If you write such a list in a straightforward manner, then it will mean what you expect it to. However, you might wind up doing something weird through a typing error. So, it's a good idea to know what the exact syntax rules are for these lists. I give the rules here for conjunction lists; the rules for disjunction lists are analogous.

A conjunction list is an expression that begins with  $\wedge$ , which is typed as  $/\wedge$ . Let  $c$  be the column in which the  $/$  occurs. The conjunction list consists of a sequence of conjuncts, each beginning with a  $\wedge$ . A conjunct is ended by any one of the following that occurs after the  $/\wedge$ :

1. Another  $/\wedge$  whose  $/$  character is in column  $c$  and is the first nonspace character on the line.
2. Any nonspace character in column  $c$  or a column to the left of column  $c$ .
3. A right delimiter whose matching left delimiter occurs before the beginning of the conjunction list. Delimiter pairs are ( ), [ ], { }, and  $\langle \rangle$ .
4. The beginning of the next module unit. (Module units are produced by the *Unit* nonterminal in the BNF grammar; they include definition and declaration statements.)

In case 1, the  $/\wedge$  begins the next conjunct in the same conjunction list. In the other three cases, the end of the conjunct is the end of the entire conjunction list. In all cases, the character ending the conjunct does not belong to the conjunct. With these rules, indentation properly delimits expressions in a conjunction list—for example:

|                           |       |                           |
|---------------------------|-------|---------------------------|
| $/\wedge$ IF $e$ THEN $P$ | means | $\wedge$ (IF $e$ THEN $P$ |
| ELSE $Q$                  |       | ELSE $Q$ )                |
| $/\wedge R$               |       | $\wedge R$                |

It's best to indent each conjunction completely to the right of its  $\wedge$  symbol. These examples illustrate precisely what happens if you don't:

|                 |                 |                 |                   |
|-----------------|-----------------|-----------------|-------------------|
| $\wedge x'$     | $\wedge x' = y$ | $\wedge x'$     | $((\wedge x'))$   |
| $= y$           | means           | $\wedge y' = x$ | $= y)$            |
| $\wedge y' = x$ |                 | $\wedge y' = x$ | $\wedge (y' = x)$ |

In the second example,  $\wedge x'$  is interpreted as a conjunction list containing only one conjunct, and the second  $\wedge$  is interpreted as an infix operator.

You can't use parentheses to circumvent the indentation rules. For example, this is illegal:

```
 $\wedge (x'$
 $= y)$
 $\wedge y' = x$
```

The rules imply that the first  $\wedge$  begins a conjunction list that is ended before the  $=$ . That conjunction list is therefore  $\wedge (x'$ , which has an unmatched left parenthesis.

The conjunction/disjunction list notation is quite robust. Even if you mess up the alignment by typing one space too few or too many—something that's easy to do when the conjuncts are long—the formula is still likely to mean what you intended. Here's an example of what happens if you misalign a conjunct:

|            |                   |                                                               |
|------------|-------------------|---------------------------------------------------------------|
| $\wedge A$ | $((\wedge A)$     | The bulleted list $\wedge A$ of one conjunct; it equals $A$ . |
| $\wedge B$ | means $\wedge B)$ | This $\wedge$ is interpreted as an infix operator.            |
| $\wedge C$ | $\wedge C$        | This $\wedge$ is interpreted as an infix operator.            |

While not interpreted the way you expected, this formula is equivalent to  $A \wedge B \wedge C$ , which is what you meant in the first place.

Most keyboards contain one key that is the source of a lot of trouble: the tab key (sometimes marked on the keyboard with a right arrow). On my computer screen, I can produce

```
A ==
 $\wedge x' = 1$
 $\wedge y' = 2$
```

by beginning the second line with eight space characters and the third with one tab character. In this case, it is unspecified whether or not the two  $\wedge$  characters occur in the same column. Tab characters are an anachronism left over from the days of typewriters and of computers with memory capacity measured in kilobytes. I strongly advise you never to use them. But, if you insist on using them, here are the rules:

- A tab character is considered to be equivalent to one or more space characters, so it occupies one or more columns.
- Identical sequences of space and tab characters that occur at the beginning of a line occupy the same number of columns.

There are no other guarantees if you use tab characters.

### 15.2.3 Comments

Comments are described in Section 3.5 on page 32. A comment may appear between any two lexemes in a specification. There are two types of comments:

- A delimited comment is a string of the form “ $(* \circ s \circ *)$ ”, where  $s$  is any string in which occurrences of “ $*$ ” and “ $*$ ” are properly matched. More precisely, a delimited comment is defined inductively to be a string of the form “ $(* \circ s_1 \circ \dots \circ s_n \circ *)$ ”, where each  $s_i$  is either (i) a string containing neither the substring “ $*$ ” nor the substring “ $*$ ”, or (ii) a delimited comment. (In particular, “ $(**)$ ” is a delimited comment.)
- An end-of-line comment is a string of the form “ $\backslash*$ ”  $\circ s \circ \langle \text{LF} \rangle$ ”, where  $s$  is any string not containing an end-of-line character  $\langle \text{LF} \rangle$ .

I like to write comments as shown here:

```
BufRcv == /\ InChan!Rcv (*****)
 /\ q' = Append(q, in.val) (* Receive message from channel *)
 /\ out (* 'in' and append to tail of q. *)
 (*****)
```

Grammatically, this piece of specification has four distinct comments, the first and last consisting of the same string  $(*****)$ . But a person reading it would regard them as a single comment, spread over four lines. This kind of commenting convention is not part of the TLA<sup>+</sup> language, but it is supported by the TLATEX typesetting program, as described in Section 13.4 on page 214.

### 15.2.4 Temporal Formulas

The BNF grammar treats  $\square$  and  $\diamond$  simply as prefix operators. However, as explained in Section 8.1 (page 88), the syntax of temporal formulas places restrictions on their use. For example,  $\square(x' = x + 1)$  is not a legal formula. It's not hard to write a BNF grammar that specifies legal temporal formulas made from the temporal operators and ordinary Boolean operators like  $\neg$  and  $\wedge$ . However, such a BNF grammar won't tell you which of these two expressions is legal:

|                                             |                                             |
|---------------------------------------------|---------------------------------------------|
| LET $F(P, Q) \triangleq P \wedge \square Q$ | LET $F(P, Q) \triangleq P \wedge \square Q$ |
| IN $F(x = 1, x = y + 1)$                    | IN $F(x = 1, x' = y + 1)$                   |

The first is legal; the second isn't because it represents the illegal formula

|                                      |                          |
|--------------------------------------|--------------------------|
| $(x = 1) \wedge \square(x' = y + 1)$ | This formula is illegal. |
|--------------------------------------|--------------------------|

The precise rules for determining if a temporal formula is syntactically well-formed involve first replacing all defined operators by their definitions, using the procedure described in Section 17.4 below. I won't bother specifying these rules.

In practice, temporal operators are not used very much in TLA<sup>+</sup> specifications, and one rarely writes definitions of new ones such as

$$F(P, Q) \triangleq P \wedge \square Q$$

The syntactic rules for expressions involving such operators are of academic interest only.

### 15.2.5 Two Anomalies

There are two sources of potential ambiguity in the grammar of TLA<sup>+</sup> that you are unlikely to encounter and that have *ad hoc* resolutions. The first of these arises from the use of  $-$  as both an infix operator (as in  $2 - 2$ ) and a prefix operator (as in  $2 + -2$ ). This poses no problem when  $-$  is used in an ordinary expression. However, there are two places in which an operator can appear by itself:

- As the argument of a higher-order operator, as in  $HOp(+, -)$ .
- In an INSTANCE substitution, such as

INSTANCE  $M$  WITH  $Plus \leftarrow +$ ,  $Minus \leftarrow -$

In both these cases, the symbol  $-$  is interpreted as the infix operator. You must type  $-.$  to denote the prefix operator. You also have to type  $-.$  if you should ever want to define the prefix  $-$  operator, as in

$$-. a \triangleq UMinus(a)$$

In ordinary expressions, you just type  $-$  as usual for both operators.

The second source of ambiguity in the TLA<sup>+</sup> syntax is an unlikely expression of the form  $\{x \in S : y \in T\}$ , which might be taken to mean either of the following:

$$\text{LET } p(x) \triangleq y \in T \text{ IN } \{x \in S : p(x)\} \quad \text{This is a subset of } S.$$

$$\text{LET } p(y) \triangleq x \in S \text{ IN } \{p(y) : y \in T\} \quad \text{This is a subset of BOOLEAN (the set \{TRUE, FALSE\}).}$$

It is interpreted as the first formula.

## 15.3 The Lexemes of TLA<sup>+</sup>

So far, this chapter has described the sequences of lexemes that form syntactically correct TLA<sup>+</sup> modules. More precisely, because of the alignment rules, syntactic correctness depends not just on the sequence of lexemes, but also on the position of each lexeme—that is, on the row and columns in which the characters of the lexeme appear. To complete the definition of the syntax of TLA<sup>+</sup>,

this section explains how a sequence of characters is turned into a sequence of lexemes.

All characters that precede the beginning of the module are ignored. Ignoring a character does not change the row or column of any other character in the sequence. The module begins with a sequence of four or more dashes (“-” characters), followed by zero or more space characters, followed by the six-character string “MODULE”. (This sequence of characters yields the first two lexemes of the module.) The remaining sequence of characters is then converted to a sequence of lexemes by iteratively applying the following rule until the module-ending == · · · == token is found:

The next lexeme begins at the next text character that is not part of a comment, and consists of the largest sequence of consecutive characters that form a legal TLA<sup>+</sup> lexeme. (It is an error if no such lexeme exists.)

Space, tab, and the end-of-line character are not text characters. It is undefined whether characters such as form feed are considered text characters. (You should not use such characters outside comments.)

In the BNF grammar, a *Name* is a lexeme that can be used as the name of a record field. The semantics of TLA<sup>+</sup>, in which *r.c* is an abbreviation for *r*[“c”], would allow any string to be a *Name*. However, some restriction is needed—for example, allowing a string like “a+b” to be a *Name* would make it impossible in practice to decide if *r.a+b* meant *r*[“a+b”] or *r*[“a”] + *b*. The one unusual restriction in the definition of *Name* on page 277 is the exclusion of strings beginning with (but not consisting entirely of) “WF\_” and “SF\_”. With this restriction, such strings are not legal TLA<sup>+</sup> lexemes. Hence, the input WF\_x(A) is broken into the five lexemes “WF\_”, “x”, “(”, “A”, and “)”, and it is interpreted as the expression WF<sub>x</sub>(A).

# Chapter 16

## The Operators of TLA<sup>+</sup>

This chapter describes the built-in operators of TLA<sup>+</sup>. Most of these operators have been described in Part I. Here, you can find brief explanations of the operators, along with references to the longer descriptions in Part I. The explanations cover some subtle points not mentioned elsewhere. The chapter can serve as a reference manual for readers who have finished Part I or who are already familiar enough with the mathematical concepts that the brief explanations are all they need.

The chapter includes a formal semantics of the operators. The rigorous description of TLA<sup>+</sup> that a formal semantics provides is usually needed only by people building TLA<sup>+</sup> tools. If you’re not building a tool and don’t have a special fondness for formalism, you will probably want to skip all the subsections titled *Formal Semantics*. However, you may some day encounter an obscure question about the meaning of a TLA<sup>+</sup> operator that is answered only by the formal semantics.

This chapter also defines some of the “semantic” conditions on the syntax of TLA<sup>+</sup> that are omitted from the grammar of Chapter 15. For example, it tells you that  $[a : \text{Nat}, a : \text{BOOLEAN}]$  is an illegal expression. Other semantic conditions on expressions arise from a combination of the definitions in this chapter and the conditions stated in Chapter 17. For example, this chapter defines  $\exists x, x : p$  to equal  $\exists x : (\exists x : p)$ , and Chapter 17 tells you that the latter expression is illegal.

### 16.1 Constant Operators

We first define the constant operators of TLA<sup>+</sup>. These are the operators of ordinary mathematics, having nothing to do with TLA or temporal logic. All

the constant operators of TLA<sup>+</sup> are listed in Table 1 on page 268 and Table 2 on page 269.

An operator combines one or more expressions into a “larger” expression. For example, the set union operator  $\cup$  combines two expressions  $e_1$  and  $e_2$  into the expression  $e_1 \cup e_2$ . Some operators don’t have simple names like  $\cup$ . There’s no name for the operator that combines the  $n$  expressions  $e_1, \dots, e_n$  to form the expression  $\{e_1, \dots, e_n\}$ . We could name it  $\{\dots\}$  or  $\{\_, \dots, \_\}$ , but that would be awkward. Instead of explicitly mentioning the operator, I’ll refer to the *construct*  $\{e_1, \dots, e_n\}$ . The distinction between an operator like  $\cup$  and the nameless one used in the construct  $\{e_1, \dots, e_n\}$  is purely syntactic, with no mathematical significance. In Chapter 17, we’ll abstract away from this syntactic difference and treat all operators uniformly. For now, we’ll stay closer to the syntax.

## Formal Semantics

A formal semantics for a language is a translation from that language into some form of mathematics. We assign a mathematical expression  $\llbracket e \rrbracket$ , called the *meaning* of  $e$ , to certain terms  $e$  in the language. Since we presumably understand the mathematics, we know what  $\llbracket e \rrbracket$  means, and that tells us what  $e$  means.

Meaning is generally defined inductively. For example, the meaning  $\llbracket e_1 \cup e_2 \rrbracket$  of the expression  $e_1 \cup e_2$  would be defined in terms of the meanings  $\llbracket e_1 \rrbracket$  and  $\llbracket e_2 \rrbracket$  of its subexpressions. This definition is said to define the semantics of the operator  $\cup$ .

Because much of TLA<sup>+</sup> is a language for expressing ordinary mathematics, much of its semantics is trivial. For example, the semantics of  $\cup$  can be defined by

$$\llbracket e_1 \cup e_2 \rrbracket \triangleq \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket$$

In this definition, the  $\cup$  to the left of the  $\triangleq$  is the TLA<sup>+</sup> symbol, while the one to the right is the set-union operator of ordinary mathematics. We could make the distinction between the two uses of the symbol  $\cup$  more obvious by writing

$$\llbracket e_1 \setminus \cup e_2 \rrbracket \triangleq \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket$$

But that wouldn’t make the definition any less trivial.

Instead of trying to maintain a distinction between the TLA<sup>+</sup> operator  $\cup$  and the operator of set theory that’s written the same, we simply use TLA<sup>+</sup> as the language of mathematics in which to define the semantics of TLA<sup>+</sup>. That is, we take as primitive certain TLA<sup>+</sup> operators that, like  $\cup$ , correspond to well-known mathematical operators. We describe the formal semantics of the constant operators of TLA<sup>+</sup> by defining them in terms of these primitive operators. We also describe the semantics of some of the primitive operators by stating the axioms that they satisfy.

### 16.1.1 Boolean Operators

The truth values of logic are written in  $\text{TLA}^+$  as `TRUE` and `FALSE`. The built-in constant `BOOLEAN` is the set consisting of those two values:

$$\text{BOOLEAN} \triangleq \{\text{TRUE}, \text{FALSE}\}$$

$\text{TLA}^+$  provides the usual operators<sup>1</sup> of propositional logic:

$$\wedge \quad \vee \quad \neg \quad \Rightarrow \text{ (implication)} \quad \equiv \quad \text{TRUE} \quad \text{FALSE}$$

They are explained in Section 1.1. Conjunctions and disjunctions can also be written as aligned lists:

$$\begin{array}{c} \wedge \ p_1 \\ \vdots \triangleq p_1 \wedge \dots \wedge p_n \\ \wedge \ p_n \end{array} \qquad \begin{array}{c} \vee \ p_1 \\ \vdots \triangleq p_1 \vee \dots \vee p_n \\ \vee \ p_n \end{array}$$

The standard quantified formulas of predicate logic are written in  $\text{TLA}^+$  as

$$\forall x : p \quad \exists x : p$$

I call these the *unbounded* quantifier constructions. The *bounded* versions are written as

$$\forall x \in S : p \quad \exists x \in S : p$$

The meanings of these expressions are described in Section 1.3.  $\text{TLA}^+$  allows some common abbreviations—for example:

$$\begin{aligned} \forall x, y : p &\triangleq \forall x : (\forall y : p) \\ \exists x, y \in S, z \in T : p &\triangleq \exists x \in S : (\exists y \in S : (\exists z \in T : p)) \end{aligned}$$

$\text{TLA}^+$  also allows bounded quantification over tuples, such as

$$\forall \langle x, y \rangle \in S : p$$

This formula is true iff, for any pair  $\langle a, b \rangle$  in  $S$ , the formula obtained from  $p$  by substituting  $a$  for  $x$  and  $b$  for  $y$  is true.

## Formal Semantics

Propositional and predicate logic, along with set theory, form the foundation of ordinary mathematics. In defining the semantics of  $\text{TLA}^+$ , we therefore take as primitives the operators of propositional logic and the simple unbounded quantifier constructs  $\exists x : p$  and  $\forall x : p$ , where  $x$  is an identifier. Among the

---

<sup>1</sup>`TRUE` and `FALSE` are operators that take no arguments.

Boolean operators described above, this leaves only the general forms of the quantifiers, given by the BNF grammar of Chapter 15, whose meanings must be defined. This is done by defining those general forms in terms of the simple forms.

The unbounded operators have the general forms

$$\forall x_1, \dots, x_n : p \quad \exists x_1, \dots, x_n : p$$

where each  $x_i$  is an identifier. They are defined in terms of quantification over a single variable by

$$\forall x_1, \dots, x_n : p \triangleq \forall x_1 : (\forall x_2 : (\dots \forall x_n : p) \dots)$$

and similarly for  $\exists$ . The bounded operators have the general forms

$$\forall \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n : p \quad \exists \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n : p$$

where each  $\mathbf{y}_i$  has the form  $x_1, \dots, x_k$  or  $\langle x_1, \dots, x_k \rangle$ , and each  $x_j$  is an identifier. The general forms of  $\forall$  are defined inductively by

$$\forall x_1, \dots, x_k \in S : p \triangleq \forall x_1, \dots, x_k : (x_1 \in S) \wedge \dots \wedge (x_k \in S) \Rightarrow p$$

$$\forall \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n : p \triangleq \forall \mathbf{y}_1 \in S_1 : \dots \forall \mathbf{y}_n \in S_n : p$$

$$\forall \langle x_1, \dots, x_k \rangle \in S : p \triangleq \forall x_1, \dots, x_k : (\langle x_1, \dots, x_k \rangle \in S) \Rightarrow p$$

where the  $\mathbf{y}_i$  are as before. In these expressions,  $S$  and the  $S_i$  lie outside the scope of the quantifier's bound identifiers. The definitions for  $\exists$  are similar. In particular:

$$\exists \langle x_1, \dots, x_k \rangle \in S : p \triangleq \exists x_1, \dots, x_k : (\langle x_1, \dots, x_k \rangle \in S) \wedge p$$

See Section 16.1.9 for further details about tuples.

### 16.1.2 The Choose Operator

A simple unbounded CHOOSE expression has the form

$$\text{CHOOSE } x : p$$

As explained in Section 6.6, the value of this expression is some arbitrary value  $v$  such that  $p$  is true if  $v$  is substituted for  $x$ , if such a  $v$  exists. If no such  $v$  exists, then the expression has a completely arbitrary value.

The bounded form of the CHOOSE expression is

$$\text{CHOOSE } x \in S : p$$

It is defined in terms of the unbounded form by

$$(16.1) \text{ CHOOSE } x \in S : p \triangleq \text{ CHOOSE } x : (x \in S) \wedge p$$

It is equal to some arbitrary value  $v$  in  $S$  such that  $p$ , with  $v$  substituted for  $x$ , is true—if such a  $v$  exists. If no such  $v$  exists, the CHOOSE expression has a completely arbitrary value.

A CHOOSE expression can also be used to choose a tuple. For example,

$$\text{ CHOOSE } \langle x, y \rangle \in S : p$$

equals some pair  $\langle v, w \rangle$  in  $S$  such that  $p$ , with  $v$  substituted for  $x$  and  $w$  substituted for  $y$ , is true—if such a pair exists. If no such pair exists, it has an arbitrary value, which need not be a pair.

The unbounded CHOOSE operator satisfies the following two rules:

$$(16.2) (\exists x : P(x)) \equiv P(\text{CHOOSE } x : P(x))$$

$$(\forall x : P(x) = Q(x)) \Rightarrow ((\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x)))$$

for any operators  $P$  and  $Q$ . We know nothing about the value chosen by CHOOSE except what we can deduce from these rules.

The second rule allows us to deduce the equality of certain CHOOSE expressions that we might expect to be different. In particular, for any operator  $P$ , if there exists no  $x$  satisfying  $P(x)$ , then  $\text{CHOOSE } x : P(x)$  equals the unique value  $\text{CHOOSE } x : \text{FALSE}$ . For example, the *Reals* module defines division by

$$a/b \triangleq \text{ CHOOSE } c \in \text{Real} : a = b * c$$

For any nonzero number  $a$ , there exists no number  $c$  such that  $a = 0 * c$ . Hence,  $a/0$  equals  $\text{CHOOSE } c : \text{FALSE}$ , for any nonzero  $a$ . We can therefore deduce that  $1/0$  equals  $2/0$ .

We would expect to be unable to deduce anything about the nonsensical expression  $1/0$ . It's a bit disquieting to prove that it equals  $2/0$ . If this upsets you, here's a way to define division that will make you happier. First define an operator *Choice* so that  $\text{Choice}(v, P)$  equals  $\text{CHOOSE } x : P(x)$  if there exists an  $x$  satisfying  $P(x)$ , and otherwise equals some arbitrary value that depends on  $v$ . There are many ways to define *Choice*; here's one:

$$\text{Choice}(v, P(\_)) \triangleq \text{ IF } \exists x : P(x) \text{ THEN } \text{CHOOSE } x : P(x)$$

$$\text{ ELSE } (\text{CHOOSE } x : x.a = v).b$$

You can then define division by

$$a/b \triangleq \text{ LET } P(c) \triangleq (c \in \text{Real}) \wedge (a = b * c)$$

$$\text{ IN } \text{Choice}(a, P)$$

This definition makes it impossible to deduce any relation between  $1/0$  and  $2/0$ . You can use *Choice* instead of CHOOSE whenever this kind of problem arises—if you consider  $1/0$  equaling  $2/0$  to be a problem. But there is seldom any practical reason for worrying about it.

## Formal Semantics

We take the construct  $\text{CHOOSE } x : p$ , where  $x$  is an identifier, to be primitive. This form of the CHOOSE operator is known to mathematicians as *Hilbert's  $\varepsilon$* . Its meaning is defined mathematically by the rules (16.2).<sup>2</sup>

An unbounded CHOOSE of a tuple is defined in terms of the simple unbounded CHOOSE construct by

$$\begin{aligned}\text{CHOOSE } \langle x_1, \dots, x_n \rangle : p &\triangleq \\ \text{CHOOSE } y : (\exists x_1, \dots, x_n : (y = \langle x_1, \dots, x_n \rangle) \wedge p)\end{aligned}$$

where  $y$  is an identifier that is different from the  $x_i$  and does not occur in  $p$ . The bounded CHOOSE construct is defined in terms of unbounded CHOOSE by (16.1), where  $x$  can be either an identifier or a tuple.

### 16.1.3 Interpretations of Boolean Operators

The meaning of a Boolean operator when applied to Boolean values is a standard part of traditional mathematics. Everyone agrees that  $\text{TRUE} \wedge \text{FALSE}$  equals  $\text{FALSE}$ . However, because TLA<sup>+</sup> is untyped, an expression like  $2 \wedge \langle 5 \rangle$  is legal. We must therefore decide what it means. There are three ways of doing this, which I call the *conservative*, *moderate*, and *liberal* interpretations.

In the conservative interpretation, the value of an expression like  $2 \wedge \langle 5 \rangle$  is completely unspecified. It could equal  $\sqrt{2}$ . It need not equal  $\langle 5 \rangle \wedge 2$ . Hence, the ordinary laws of logic, such as the commutativity of  $\wedge$ , are valid only for Boolean values.

In the liberal interpretation, the value of  $2 \wedge \langle 5 \rangle$  is specified to be a Boolean. It is not specified whether it equals  $\text{TRUE}$  or  $\text{FALSE}$ . However, all the ordinary laws of logic, such as the commutativity of  $\wedge$ , are valid. Hence,  $2 \wedge \langle 5 \rangle$  equals  $\langle 5 \rangle \wedge 2$ . More precisely, any tautology of propositional or predicate logic, such as

$$(\forall x : p) \equiv \neg(\exists x : \neg p)$$

is valid, even if  $p$  is not necessarily a Boolean for all values of  $x$ .<sup>3</sup> It is easy to show that the liberal approach is sound.<sup>4</sup> For example, one way of defining operators that satisfy the liberal interpretation is to consider any non-Boolean value to be equivalent to  $\text{FALSE}$ .

The conservative and liberal interpretations are equivalent for most specifications, except for ones that use Boolean-valued functions. In practice, the

<sup>2</sup>Hilbert's  $\varepsilon$  is discussed at length in *Mathematical Logic and Hilbert's  $\varepsilon$ -Symbol* by A. C. Leisenring, published by Gordon and Breach, New York, 1969.

<sup>3</sup>Equality ( $=$ ) is not an operator of propositional or predicate logic; this tautology need not be valid for non-Boolean values if  $\equiv$  is replaced by  $=$ .

<sup>4</sup>A sound logic is one in which  $\text{FALSE}$  is not provable.

conservative interpretation doesn't permit you to use  $f[x]$  as a Boolean expression even if  $f$  is defined to be a Boolean-valued function. For example, suppose we define the function  $tnat$  by

$$tnat \triangleq [n \in Nat \mapsto \text{TRUE}]$$

so  $tnat[n]$  equals TRUE for all  $n$  in  $Nat$ . The formula

$$(16.3) \forall n \in Nat : tnat[n]$$

equals TRUE in the liberal interpretation, but not in the conservative interpretation. Formula (16.3) is equivalent to

$$\forall n : (n \in Nat) \Rightarrow tnat[n]$$

which asserts that  $(n \in Nat) \Rightarrow tnat[n]$  is true for all  $n$ , including, for example,  $n = 1/2$ . For (16.3) to equal TRUE, the formula  $(1/2 \in Nat) \Rightarrow tnat[1/2]$ , which equals FALSE  $\Rightarrow tnat[1/2]$ , must equal TRUE. But the value of  $tnat[1/2]$  is not specified; it might equal  $\sqrt{2}$ . The formula FALSE  $\Rightarrow \sqrt{2}$  equals TRUE in the liberal interpretation; its value is unspecified in the conservative interpretation. Hence, the value of (16.3) is unspecified in the conservative interpretation. If we are using the conservative interpretation, instead of (16.3), we should write

$$\forall n \in Nat : (tnat[n] = \text{TRUE})$$

This formula equals TRUE in both interpretations.

The conservative interpretation is philosophically more satisfying, since it makes no assumptions about a silly expression like  $2 \wedge \langle 5 \rangle$ . However, as we have just seen, it would be nice if the not-so-silly formula FALSE  $\Rightarrow \sqrt{2}$  equaled TRUE. We therefore introduce the moderate interpretation, which lies between the conservative and liberal interpretations. It assumes only that expressions involving FALSE and TRUE have their expected values—for example, FALSE  $\Rightarrow \sqrt{2}$  equals TRUE, and FALSE  $\wedge 2$  equals FALSE. In the moderate interpretation, (16.3) equals TRUE, but the value of  $\langle 5 \rangle \wedge 2$  is still completely unspecified.

The laws of logic still do not hold unconditionally in the moderate interpretation. The formulas  $p \wedge q$  and  $q \wedge p$  are equivalent only if  $p$  and  $q$  are both Booleans, or if one of them equals FALSE. When using the moderate interpretation, we still have to check that all the relevant values are Booleans before applying any of the ordinary rules of logic in a proof. This can be burdensome in practice.

The semantics of TLA<sup>+</sup> asserts that the rules of the moderate interpretation are valid. The liberal interpretation is neither required nor forbidden. You should write specifications that make sense under the moderate interpretation. However, you (and the implementer of a tool) are free to use the liberal interpretation if you wish.

### 16.1.4 Conditional Constructs

TLA<sup>+</sup> provides two conditional constructs for forming expressions that are inspired by constructs from programming languages: IF/THEN/ELSE and CASE.

The IF/THEN/ELSE construct was introduced on page 16 of Section 2.2. Its general form is

IF  $p$  THEN  $e_1$  ELSE  $e_2$

It equals  $e_1$  if  $p$  is true, and  $e_2$  if  $p$  is false.

An expression can sometimes be simplified by using a CASE construct instead of nested IF/THEN/ELSE constructs. The CASE construct has two general forms:

(16.4) CASE  $p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n$

CASE  $p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n \sqcap \text{OTHER} \rightarrow e$

If some  $p_i$  is true, then the value of these expressions is some  $e_i$  such that  $p_i$  is true. For example, the expression

CASE  $n \geq 0 \rightarrow e_1 \sqcap n \leq 0 \rightarrow e_2$

equals  $e_1$  if  $n > 0$  is true, equals  $e_2$  if  $n < 0$  is true, and equals either  $e_1$  or  $e_2$  if  $n = 0$  is true. In the latter case, the semantics of TLA<sup>+</sup> does not specify whether the expression equals  $e_1$  or  $e_2$ . The CASE expressions (16.4) are generally used when the  $p_i$  are mutually disjoint, so at most one  $p_i$  can be true.

The two expressions (16.4) differ when  $p_i$  is false for all  $i$ . In that case, the value of the first is unspecified, while the value of the second is  $e$ , the OTHER expression. If you use a CASE expression without an OTHER clause, the value of the expression should matter only when  $\exists i \in 1 \dots n : p_i$  is true.

### Formal Semantics

The IF/THEN/ELSE and CASE constructs are defined as follows in terms of CHOOSE:

$$\begin{aligned} \text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2 &\triangleq \\ \text{CHOOSE } v : (p \Rightarrow (v = e_1)) \wedge (\neg p \Rightarrow (v = e_2)) & \\ \text{CASE } p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n &\triangleq \\ \text{CHOOSE } v : (p_1 \wedge (v = e_1)) \vee \dots \vee (p_n \wedge (v = e_n)) & \\ \text{CASE } p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n \sqcap \text{OTHER} \rightarrow e &\triangleq \\ \text{CASE } p_1 \rightarrow e_1 \sqcap \dots \sqcap p_n \rightarrow e_n \sqcap \neg(p_1 \vee \dots \vee p_n) \rightarrow e & \end{aligned}$$

### 16.1.5 The Let/In Construct

The LET/IN construct was introduced on page 60 of Section 5.6. The expression

LET  $d \triangleq f$  IN  $e$

equals  $e$  in the context of the definition  $d \triangleq f$ . For example,

LET  $sq(i) \triangleq i * i$  IN  $sq(1) + sq(2) + sq(3)$

equals  $1 * 1 + 2 * 2 + 3 * 3$ , which equals 14. The general form of the construct is

LET  $\Delta_1 \dots \Delta_n$  IN  $e$

where each  $\Delta_i$  has the syntactic form of any TLA<sup>+</sup> definition. Its value is  $e$  in the context of the definitions  $\Delta_i$ . More precisely, it equals

LET  $\Delta_1$  IN (LET  $\Delta_2$  IN ( ... LET  $\Delta_n$  IN  $e$  ) ... )

Hence, the symbol defined in  $\Delta_1$  can be used in the definitions  $\Delta_2, \dots, \Delta_n$ .

### Formal Semantics

The formal semantics of the LET construct is defined below in Section 17.4 (page 325).

### 16.1.6 The Operators of Set Theory

TLA<sup>+</sup> provides the following operators on sets:

$\in$      $\notin$      $\cup$      $\cap$      $\subseteq$      $\setminus$     UNION    SUBSET

and the following set constructors:

$\{e_1, \dots, e_n\}$      $\{x \in S : p\}$      $\{e : x \in S\}$

They are all described in Section 1.2 (page 11) and Section 6.1 (page 65). Equality is also an operator of set theory, since it formally means equality of sets. TLA<sup>+</sup> provides the usual operators  $=$  and  $\neq$ .

The set construct  $\{x \in S : p\}$  can also be used with  $x$  a tuple of identifiers. For example,

$\{\langle a, b \rangle \in Nat \times Nat : a > b\}$

is the set of all pairs of natural numbers whose first component is greater than its second—pairs such as  $\langle 3, 1 \rangle$ . In the set construct  $\{e : x \in S\}$ , the clause  $x \in S$  can be generalized in exactly the same way as in a bounded quantifier such as  $\forall x \in S : p$ . For example,

$\{\langle a, b, c \rangle : a, b \in Nat, c \in Real\}$

is the set of all triples whose first two components are natural numbers and whose third component is a real number.

## Formal Semantics

TLA<sup>+</sup> is based on Zermelo-Frnkel set theory, in which every value is a set. In set theory,  $\in$  is taken as a primitive, undefined operator. We could define all the other operators of set theory in terms of  $\in$ , using predicate logic and the CHOOSE operator. For example, set union could be defined by

$$S \cup T \triangleq \text{CHOOSE } U : \forall x : (x \in U) \equiv (x \in S) \vee (x \in T)$$

(To reason about  $\cup$ , we would need axioms from which we can deduce the existence of the chosen set  $U$ .) Another approach we could take is to let certain of the operators be primitive and define the rest in terms of them. For example,  $\cup$  can be defined in terms of UNION and the construct  $\{e_1, \dots, e_n\}$  by

$$S \cup T \triangleq \text{UNION } \{S, T\}$$

We won't try to distinguish a small set of primitive operators; instead, we treat  $\cup$  and UNION as equally primitive. Operators that we take to be primitive are defined mathematically in terms of the rules that they satisfy. For example,  $S \cup T$  is defined by

$$\forall x : (x \in (S \cup T)) \equiv (x \in S) \vee (x \in T)$$

However, there is no such defining rule for the primitive operator  $\in$ . We take only the simple forms of the constructs  $\{x \in S : p\}$  and  $\{e : x \in S\}$  as primitive, and we define the more general forms in terms of them.

$$S = T \triangleq \forall x : (x \in S) \equiv (x \in T).$$

$$e_1 \neq e_2 \triangleq \neg(e_1 = e_2).$$

$$e \notin S \triangleq \neg(e \in S).$$

$$S \cup T \text{ is defined by } \forall x : (x \in (S \cup T)) \equiv (x \in S) \vee (x \in T).$$

$$S \cap T \text{ is defined by } \forall x : (x \in (S \cap T)) \equiv (x \in S) \wedge (x \in T).$$

$$S \subseteq T \triangleq \forall x : (x \in S) \Rightarrow (x \in T).$$

$$S \setminus T \text{ is defined by } \forall x : (x \in (S \setminus T)) \equiv (x \in S) \wedge (x \notin T).$$

$$\text{SUBSET } S \text{ is defined by } \forall T : (T \in \text{SUBSET } S) \equiv (T \subseteq S).$$

$$\text{UNION } S \text{ is defined by } \forall x : (x \in \text{UNION } S) \equiv (\exists T \in S : x \in T).$$

$$\{e_1, \dots, e_n\} \triangleq \{e_1\} \cup \dots \cup \{e_n\},$$

where  $\{e\}$  is defined by

$$\forall x : (x \in \{e\}) \equiv (x = e)$$

For  $n = 0$ , this construct is the empty set  $\{\}$ , defined by

$$\forall x : x \notin \{\}$$

$\{x \in S : p\}$

where  $x$  is a bound identifier or a tuple of bound identifiers. The expression  $S$  is outside the scope of the bound identifier(s). For  $x$  an identifier, this is a primitive expression that is defined mathematically by

$$\forall y : (y \in \{x \in S : p\}) \equiv (y \in S) \wedge \hat{p}$$

where the identifier  $y$  does not occur in  $S$  or  $p$ , and  $\hat{p}$  is  $p$  with  $y$  substituted for  $x$ . For  $x$  a tuple, the expression is defined by

$$\begin{aligned} \{\langle x_1, \dots, x_n \rangle \in S : p\} &\triangleq \\ \{y \in S : (\exists x_1, \dots, x_n : (y = \langle x_1, \dots, x_n \rangle) \wedge p)\} \end{aligned}$$

where  $y$  is an identifier different from the  $x_i$  that does not occur in  $S$  or  $p$ . See Section 16.1.9 for further details about tuples.

$\{e : \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n\}$

where each  $\mathbf{y}_i$  has the form  $x_1, \dots, x_k$  or  $\langle x_1, \dots, x_k \rangle$ , and each  $x_j$  is an identifier that is bound in the expression. The expressions  $S_i$  lie outside the scope of the bound identifiers. The simple form  $\{e : x \in S\}$ , for  $x$  an identifier, is taken to be primitive and is defined by

$$\forall y : (y \in \{e : x \in S\}) \equiv (\exists x \in S : e = y)$$

The general form is defined inductively in terms of the simple form by

$$\begin{aligned} \{e : \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n\} &\triangleq \\ \text{UNION } \{\{e : \mathbf{y}_1 \in S_1, \dots, \mathbf{y}_{n-1} \in S_{n-1}\} : \mathbf{y}_n \in S_n\} \\ \{e : x_1, \dots, x_n \in S\} &\triangleq \{e : x_1 \in S, \dots, x_n \in S\} \\ \{e : \langle x_1, \dots, x_n \rangle \in S\} &\triangleq \\ \{(\text{LET } z \triangleq \text{CHOOSE } \langle x_1, \dots, x_n \rangle : y = \langle x_1, \dots, x_n \rangle \\ x_1 &\triangleq z[1] \\ &\vdots \\ x_n &\triangleq z[n] \text{ IN } e) : y \in S\} \end{aligned}$$

where the  $x_i$  are identifiers, and  $y$  and  $z$  are identifiers distinct from the  $x_i$  that do not occur in  $e$  or  $S$ . See Section 16.1.9 for further details about tuples.

### 16.1.7 Functions

Functions are described in Section 5.2 (page 48); the difference between functions and operators is discussed in Section 6.4 (page 69). In TLA<sup>+</sup>, we write  $f[v]$  for

the value of the function  $f$  applied to  $v$ . A function  $f$  has a domain  $\text{DOMAIN } f$ , and the value of  $f[v]$  is specified only if  $v$  is an element of  $\text{DOMAIN } f$ . We let  $[S \rightarrow T]$  denote the set of all functions  $f$  such that  $\text{DOMAIN } f = S$  and  $f[v] \in T$ , for all  $v \in S$ .

Functions can be described explicitly with the construct

$$(16.5) [x \in S \mapsto e]$$

This is the function  $f$  with domain  $S$  such that  $f[v]$  equals the value obtained by substituting  $v$  for  $x$  in  $e$ , for any  $v \in S$ . For example,

$$[n \in \text{Nat} \mapsto 1/(n + 1)]$$

is the function  $f$  with domain  $\text{Nat}$  such that  $f[0] = 1$ ,  $f[1] = 1/2$ ,  $f[2] = 1/3$ , etc. We can define an identifier  $fcn$  to equal the function (16.5) by writing

$$(16.6) fcn[x \in S] \triangleq e$$

The identifier  $fcn$  can appear in the expression  $e$ , in which case this is a recursive function definition. Recursive function definitions were introduced in Section 5.5 (page 54) and discussed in Section 6.3 (page 67).

The EXCEPT construct describes a function that is “almost the same as” another function. For example,

$$(16.7) [f \text{ EXCEPT } ![u] = a, !v] = b]$$

is the function  $\tilde{f}$  that is the same as  $f$ , except that  $\tilde{f}[u] = a$  and  $\tilde{f}[v] = b$ . More precisely, (16.7) equals

$$[x \in \text{DOMAIN } f \mapsto \text{IF } x = v \text{ THEN } b \\ \text{ELSE IF } x = u \text{ THEN } a \text{ ELSE } f[x]]$$

Hence, if neither  $u$  nor  $v$  is in the domain of  $f$ , then (16.7) equals  $f$ . If  $u = v$ , then (16.7) equals  $[f \text{ EXCEPT } !v] = b$ .

An exception clause can have the general form  $![v_1] \cdots [v_n] = e$ . For example,

$$(16.8) [f \text{ EXCEPT } !u[v] = a]$$

is the function  $\tilde{f}$  that is the same as  $f$ , except that  $\tilde{f}[u][v]$  equals  $a$ . That is,  $\tilde{f}$  is the same as  $f$ , except that  $\tilde{f}[u]$  is the function that is the same as  $f[u]$ , except that  $\tilde{f}[u][v] = a$ . The symbol  $@$  occurring in an exception clause stands for the “original value”. For example, an  $@$  in the expression  $a$  of (16.8) denotes  $f[u][v]$ .

In TLA<sup>+</sup>, a function of multiple arguments is one whose domain is a set of tuples; and  $f[v_1, \dots, v_n]$  is an abbreviation for  $f[\langle v_1, \dots, v_n \rangle]$ . The  $x \in S$  clause (16.5) and (16.6) can be generalized in the same way as in a bounded quantifier—for example, here are two different ways of writing the same function:

$$[m, n \in \text{Nat}, r \in \text{Real} \mapsto e] \quad [\langle m, n, r \rangle \in \text{Nat} \times \text{Nat} \times \text{Real} \mapsto e]$$

This is a function whose domain is a set of triples. It is not the same as the function

$$[\langle m, n \rangle \in Nat \times Nat, r \in Real \mapsto e]$$

whose domain is the set  $(Nat \times Nat) \times Real$  of pairs like  $\langle \langle 1, 3 \rangle, 1/3 \rangle$ , whose first element is a pair of natural numbers.

## Formal Semantics

Mathematicians traditionally define a function to be a set of pairs. In TLA<sup>+</sup>, pairs (and all tuples) are functions. We take as primitives the constructs

$$f[e] \quad \text{DOMAIN } f \quad [S \rightarrow T] \quad [x \in S \mapsto e]$$

where  $x$  is an identifier. These constructs are defined mathematically by the rules they satisfy. The other constructs, and the general forms of the construct  $[x \in S \mapsto e]$ , are defined in terms of them. These definitions use the operator  $IsAFcn$ , which is defined as follows so that  $IsAFcn(f)$  is true iff  $f$  is a function:

$$IsAFcn(f) \triangleq f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

The first rule, which is not naturally associated with any one construct, is that two functions are equal iff they have the same domain and assign the same value to each element in their domain:

$$\begin{aligned} \forall f, g : IsAFcn(f) \wedge IsAFcn(g) \Rightarrow \\ ((f = g) \equiv \wedge \text{DOMAIN } f = \text{DOMAIN } g \\ \wedge \forall x \in \text{DOMAIN } f : f[x] = g[x]) \end{aligned}$$

The rest of the semantics of functions is given below. There is no separate defining rule for the DOMAIN operator.

$$f[e_1, \dots, e_n]$$

where the  $e_i$  are expressions. For  $n = 1$ , this is a primitive expression.

For  $n > 1$ , it is defined by

$$f[e_1, \dots, e_n] = f[\langle e_1, \dots, e_n \rangle]$$

The tuple  $\langle e_1, \dots, e_n \rangle$  is defined in Section 16.1.9.

$$[\mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n \mapsto e]$$

where each  $\mathbf{y}_i$  has the form  $x_1, \dots, x_k$  or  $\langle x_1, \dots, x_k \rangle$ , and each  $x_j$  is an identifier that is bound in the expression. The expressions  $S_i$  lie outside the scope of the bound identifiers. The simple form  $[x \in S \mapsto e]$ , for  $x$  an identifier, is primitive and is defined by two rules:

$$(\text{DOMAIN } [x \in S \mapsto e]) = S$$

$$\forall y \in S : [x \in S \mapsto e][y] = \text{LET } x \triangleq y \text{ IN } e$$

where  $y$  is an identifier different from  $x$  that does not occur in  $S$  or  $e$ . The general form of the construct is defined inductively in terms of the simple form by

$$\begin{aligned}
 [x_1 \in S_1, \dots, x_n \in S_n \mapsto e] &\triangleq [\langle x_1, \dots, x_n \rangle \in S_1 \times \dots \times S_n \mapsto e] \\
 [\dots, x_1, \dots, x_k \in S_i, \dots \mapsto e] &\triangleq [\dots, x_1 \in S_i, \dots, x_k \in S_i, \dots \mapsto e] \\
 [\dots, \langle x_1, \dots, x_k \rangle \in S_i, \dots \mapsto e] &\triangleq \\
 [\dots, y \in S_i, \dots \mapsto \text{LET } z &\triangleq \text{CHOOSE } \langle x_1, \dots, x_k \rangle : y = \langle x_1, \dots, x_k \rangle \\
 &\quad x_1 \triangleq z[1] \\
 &\quad \vdots \\
 &\quad x_k \triangleq z[k] \text{ IN } e]
 \end{aligned}$$

where  $y$  and  $z$  are identifiers that do not appear anywhere in the original expression. See Section 16.1.9 for details about tuples.

$[S \rightarrow T]$  is defined by

$$\begin{aligned}
 \forall f : f \in [S \rightarrow T] &\equiv \\
 \text{IsAFcn}(f) \wedge (S = \text{DOMAIN } f) \wedge (\forall x \in S : f[x] \in T)
 \end{aligned}$$

where  $x$  and  $f$  do not occur in  $S$  or  $T$ , and  $\text{IsAFcn}$  is defined above.

$[f \text{ EXCEPT } !\mathbf{a}_1 = e_1, \dots, !\mathbf{a}_n = e_n]$

where each  $\mathbf{a}_i$  has the form  $[d_1] \dots [d_k]$  and each  $d_j$  is an expression. For the simple case when  $n = 1$  and  $\mathbf{a}_1$  is  $[d]$ , this is defined by<sup>5</sup>

$$\begin{aligned}
 [f \text{ EXCEPT } !(d) = e] &\triangleq \\
 [y \in \text{DOMAIN } f \mapsto \text{IF } y = d \text{ THEN LET } @ \triangleq f[d] \text{ IN } e \\
 &\quad \text{ELSE } f[y]]
 \end{aligned}$$

where  $y$  does not occur in  $f$ ,  $d$ , or  $e$ . The general form is defined inductively in terms of this simple case by

$$\begin{aligned}
 [f \text{ EXCEPT } !\mathbf{a}_1 = e_1, \dots, !\mathbf{a}_n = e_n] &\triangleq \\
 [[f \text{ EXCEPT } !\mathbf{a}_1 = e_1, \dots, !\mathbf{a}_{n-1} = e_{n-1}] \text{ EXCEPT } !\mathbf{a}_n = e_n] \\
 [f \text{ EXCEPT } !(d_1) \dots (d_k) = e] &\triangleq \\
 [f \text{ EXCEPT } !(d_1) = [@ \text{ EXCEPT } !(d_2) \dots (d_k) = e]]
 \end{aligned}$$

$f[\mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n] \triangleq e$  is defined to be an abbreviation for

$$f \triangleq \text{CHOOSE } f : f = [\mathbf{y}_1 \in S_1, \dots, \mathbf{y}_n \in S_n \mapsto e]$$

---

<sup>5</sup>Since  $@$  is not actually an identifier,  $\text{LET } @ \triangleq \dots$  isn't legal TLA<sup>+</sup> syntax. However, its meaning should be clear.

### 16.1.8 Records

TLA<sup>+</sup> borrows from programming languages the concept of a record. Records were introduced in Section 3.2 (page 28) and further explained in Section 5.2 (page 48). As in programming languages,  $r.h$  is the  $h$  field of record  $r$ . Records can be written explicitly as

$$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$$

which equals the record with  $n$  fields, whose  $h_i$  field equals  $e_i$ , for  $i = 1, \dots, n$ . The expression

$$[h_1 : S_1, \dots, h_n : S_n]$$

is the set of all such records with  $e_i \in S_i$ , for  $i = 1, \dots, n$ . These expressions are legal only if the  $h_i$  are all different. For example,  $[a : S, a : T]$  is illegal.

The EXCEPT construct, explained in Section 16.1.7 above, can be used for records as well as functions. For example,

$$[r \text{ EXCEPT } !.a = e]$$

is the record  $\hat{r}$  that is the same as  $r$ , except that  $\hat{r}.a = e$ . An exception clause can mix function application and record fields. For example,

$$[f \text{ EXCEPT } ![v].a = e]$$

is the function  $\hat{f}$  that is the same as  $f$ , except that  $\hat{f}[v].a = e$ .

In TLA<sup>+</sup>, a record is a function whose domain is a finite set of strings, where  $r.h$  means  $r["h"]$ , for any expression  $r$  and record field  $h$ . Thus, the following two expressions describe the same record:

$$[fo \mapsto 7, ba \mapsto 8] \quad [x \in \{"fo", "ba"\} \mapsto \text{IF } x = "fo" \text{ THEN } 7 \text{ ELSE } 8]$$

The name of a record field is syntactically an identifier. In the ASCII version of TLA<sup>+</sup>, it is a string of letters, digits, and the underscore character ( $_$ ) that contains at least one letter. Strings are described below in Section 16.1.10.

## Formal Semantics

The record constructs are defined in terms of function constructs.

$$e.h \triangleq e["h"]$$

$$[h_1 \mapsto e_1, \dots, h_n \mapsto e_n] \triangleq$$

$$[y \in \{h_1, \dots, h_n\} \mapsto$$

$$\text{CASE } (y = "h_1") \rightarrow e_1 \square \dots \square (y = "h_n") \rightarrow e_n]$$

where  $y$  does not occur in any of the expressions  $e_i$ . The  $h_i$  must all be distinct.

$$[h_1 : S_1, \dots, h_n : S_n] \triangleq \{ [h_1 \mapsto y_1, \dots, h_n \mapsto y_n] : y_1 \in S_1, \dots, y_n \in S_n \}$$

where the  $y_i$  do not occur in any of the expressions  $S_j$ . The  $h_i$  must all be distinct.

$$[r \text{ EXCEPT } !\mathbf{a}_1 = e_1, \dots, !\mathbf{a}_n = e_n]$$

where  $\mathbf{a}_i$  has the form  $b_1 \dots b_k$  and each  $b_j$  is either (i)  $[d]$ , where  $d$  is an expression, or (ii)  $.h$ , where  $h$  is a record field. It is defined to equal the corresponding function EXCEPT construct in which each  $.h$  is replaced by  $[^h]$ .

### 16.1.9 Tuples

An  $n$ -tuple is written in TLA<sup>+</sup> as  $\langle e_1, \dots, e_n \rangle$ . As explained in Section 5.4, an  $n$ -tuple is defined to be a function whose domain is the set  $\{1, \dots, n\}$ , where  $\langle e_1, \dots, e_n \rangle[i] = e_i$ , for  $1 \leq i \leq n$ . The Cartesian product  $S_1 \times \dots \times S_n$  is the set of all  $n$ -tuples  $\langle e_1, \dots, e_n \rangle$  such that  $e_i \in S_i$ , for  $1 \leq i \leq n$ .

In TLA<sup>+</sup>,  $\times$  is not an associative operator. For example,

$$\begin{aligned} \langle 1, 2, 3 \rangle &\in \text{Nat} \times \text{Nat} \times \text{Nat} \\ \langle \langle 1, 2 \rangle, 3 \rangle &\in (\text{Nat} \times \text{Nat}) \times \text{Nat} \\ \langle 1, \langle 2, 3 \rangle \rangle &\in \text{Nat} \times (\text{Nat} \times \text{Nat}) \end{aligned}$$

and the tuples  $\langle 1, 2, 3 \rangle$ ,  $\langle \langle 1, 2 \rangle, 3 \rangle$ , and  $\langle 1, \langle 2, 3 \rangle \rangle$  are not equal. More precisely, the triple  $\langle 1, 2, 3 \rangle$  is unequal to either of the pairs  $\langle \langle 1, 2 \rangle, 3 \rangle$  or  $\langle 1, \langle 2, 3 \rangle \rangle$  because a triple and a pair have unequal domains. The semantics of TLA<sup>+</sup> does not specify if  $\langle 1, 2 \rangle$  equals 1 or if 3 equals  $\langle 2, 3 \rangle$ , so we don't know whether or not  $\langle \langle 1, 2 \rangle, 3 \rangle$  and  $\langle 1, \langle 2, 3 \rangle \rangle$  are equal.

The 0-tuple  $\langle \rangle$  is the unique function having an empty domain. The 1-tuple  $\langle e \rangle$  is different from  $e$ . That is, the semantics does not specify whether or not they are equal. There is no special notation for writing a set of 1-tuples. The easiest way to denote the set of all 1-tuples  $\langle e \rangle$  with  $e \in S$  is  $\{\langle e \rangle : e \in S\}$ .

In the standard *Sequences* module, described in Section 18.1 (page 339), an  $n$ -element sequence is represented as an  $n$ -tuple. The module defines several useful operators on sequences/tuples.

### Formal Semantics

Tuples and Cartesian products are defined in terms of functions (defined in Section 16.1.7) and the set *Nat* of natural numbers (defined in Section 16.1.11).

$$\langle e_1, \dots, e_n \rangle \triangleq [i \in \{j \in \text{Nat} : (1 \leq j) \wedge (j \leq n)\} \mapsto e_i]$$

where  $i$  does not occur in any of the expressions  $e_j$ .

$S_1 \times \cdots \times S_n \triangleq \{ \langle y_1, \dots, y_n \rangle : y_1 \in S_1, \dots, y_n \in S_n \}$   
 where the identifiers  $y_i$  do not occur in any of the expressions  $S_j$ .

### 16.1.10 Strings

TLA<sup>+</sup> defines a string to be a tuple of characters. (Tuples are defined in Section 16.1.9 above.) Thus, “abc” equals

$$\langle \text{“abc”}[1], \text{“abc”}[2], \text{“abc”}[3] \rangle$$

The semantics of TLA<sup>+</sup> does not specify what a character is. However, it does specify that different characters (those having different computer representations) are different. Thus “a”[1], “b”[1], and “A”[1] (the characters *a*, *b*, and *A*) are all different. The built-in operator STRING is defined to be the set of all strings.

Although TLA<sup>+</sup> doesn’t specify what a character is, it’s easy to define operators that assign values to characters. For example, here’s the definition of an operator *Ascii* that assigns to every lower-case letter its ASCII representation.<sup>6</sup>

$$\begin{aligned} \text{Ascii}(\text{char}) &\triangleq 96 + \text{CHOOSE } i \in 1 \dots 26 : \\ &\quad \text{“abcdefghijklmnopqrstuvwxyz”}[i] = \text{char} \end{aligned}$$

This defines *Ascii*(“a”[1]) to equal 97, the ASCII code for the letter *a*, and *Ascii*(“z”[1]) to equal 122, the ASCII code for *z*. Section 11.1.4 on page 179 illustrates how a specification can make use of the fact that strings are tuples.

Exactly what characters may appear in a string is system-dependent. A Japanese version of TLA<sup>+</sup> might not allow the character *a*. The standard ASCII version contains the following characters:

*a b c d e f g h i j k l m n o p q r s t u v w x y z*  
*A B C D E F G H I J K L M N O P Q R S T U V W X Y Z*  
*0 1 2 3 4 5 6 7 8 9*  
*~ @ # \$ % ^ & \* \_ - + = ( ) { } [ ] < > | / \ , . ? : ; ' "*  
*⟨HT⟩ (tab) ⟨LF⟩ (line feed) ⟨FF⟩ (form feed) ⟨CR⟩ (carriage return)*

plus the space character. Since strings are delimited by a double-quote ("), some convention is needed for typing a string that contains a double-quote. Conventions are also needed to type characters like ⟨LF⟩ within a string. In the ASCII version of TLA<sup>+</sup>, the following pairs of characters, beginning with a \ character, are used to represent these special characters:

|     |   |    |      |    |      |
|-----|---|----|------|----|------|
| \"  | " | \t | ⟨HT⟩ | \f | ⟨FF⟩ |
| \\\ | \ | \n | ⟨LF⟩ | \r | ⟨CR⟩ |

---

<sup>6</sup>This clever way of using CHOOSE to map from characters to numbers was pointed out to me by Georges Gonthier.

With this convention, "a\\\"b\\\"" represents the string consisting of the following five characters: *a* \ " *b* ". In the ASCII version of TLA<sup>+</sup>, a \ character can appear in a string expression only as the first character of one of these six two-character sequences.

## Formal Semantics

We assume a set *Char* of characters, which may depend on the version of TLA<sup>+</sup>. (The identifier *Char* is not a pre-defined symbol of TLA<sup>+</sup>.)

STRING  $\triangleq$  *Seq*(*Char*)

where *Seq* is the operator defined in the *Sequences* module of Section 18.1 so that *Seq*(*S*) is the set of all finite sequences of elements of *S*.

"*c*<sub>1</sub> ... *c*<sub>*n*</sub>"  $\triangleq$   $\langle c_1, \dots, c_n \rangle$

where each *c*<sub>*i*</sub> is some representation of a character in *Char*.

### 16.1.11 Numbers

TLA<sup>+</sup> defines a sequence of digits like 63 to be the usual natural number—that is, 63 equals  $6 * 10 + 3$ . TLA<sup>+</sup> also allows the binary representation \b111111, the octal representation \o77, and the hexadecimal representation \h3F of that number. (Case is ignored in the prefixes and in the hexadecimal representation, so \H3F and \h3f are equivalent to \h3F.) Decimal numbers are also pre-defined in TLA<sup>+</sup>; for example, 3.14159 equals 314159/10<sup>5</sup>.

Numbers are pre-defined in TLA<sup>+</sup>, so 63 is defined even in a module that does not extend or instantiate one of the standard numbers modules. However, sets of numbers like *Nat* and arithmetic operators like + are not. You can write a module that defines + any way you want, in which case  $40 + 23$  need not equal 63. Of course,  $40 + 23$  does equal 63 for + defined by the standard numbers modules *Naturals*, *Integers*, and *Reals*, which are described in Section 18.4.

## Formal Semantics

The set *Nat* of natural numbers, along with its zero element *Zero* and successor function *Succ*, is defined in module *Peano* on page 345. The meaning of a representation of a natural number is defined in the usual manner:

$$0 \triangleq \text{Zero} \quad 1 \triangleq \text{Succ}[\text{Zero}] \quad 2 \triangleq \text{Succ}[\text{Succ}[\text{Zero}]] \quad \dots$$

The *ProtoReals* module on pages 346–347 defines the set *Real* of real numbers to be a superset of the set *Nat*, and defines the usual arithmetic operators on

real numbers. The meaning of a decimal number is defined in terms of these operators by

$$c_1 \dots c_m . d_1 \dots d_n \triangleq c_1 \dots c_m d_1 \dots d_n / 10^n$$

## 16.2 Nonconstant Operators

The nonconstant operators are what distinguish TLA<sup>+</sup> from ordinary mathematics. There are two classes of nonconstant operators: action operators, listed in Table 3 on page 269, and temporal operators, listed in Table 4 on page 269.

Section 16.1 above talks about the meanings of the built-in constant operators of TLA<sup>+</sup>, without considering their arguments. We can do that for constant operators, since the meaning of  $\subseteq$  in the expression  $e_1 \subseteq e_2$  doesn't depend on whether or not the expressions  $e_1$  and  $e_2$  contain variables or primes. To understand the nonconstant operators, we need to consider their arguments. Thus, we can no longer talk about the meanings of the operators in isolation; we must describe the meanings of expressions built from those operators.

A *basic* expression is one that contains built-in TLA<sup>+</sup> operators, declared constants, and declared variables. We now describe the meaning of all basic TLA<sup>+</sup> expressions, including ones that contain nonconstant built-in operators. We start by considering basic constant expressions, ones containing only the constant operators we have already studied and declared constants.

### 16.2.1 Basic Constant Expressions

Section 16.1 above defines the meanings of the constant operators. This in turn defines the meaning of any expression built from these operators and declared constants. For example, if  $S$  and  $T$  are declared by

CONSTANTS  $S, T(\_)$

then  $\exists x : S \subseteq T(x)$  is a formula that equals TRUE if there is some value  $v$  such that every element of  $S$  is an element of  $T(v)$ , and otherwise equals FALSE. Whether  $\exists x : S \subseteq T(x)$  equals TRUE or FALSE depends on what actual values we assign to  $S$  and to  $T(v)$ , for all  $v$ ; so that's as far as we can go in assigning a meaning to the expression.

A formula is a Boolean-valued expression. There are some basic constant formulas that are true regardless of the values we assign to their declared constants—for example, the formula

$$(S \subseteq T) \equiv (S \cap T = S)$$

Such a formula is said to be *valid*.

## Formal Semantics

Section 16.1 defines all the built-in constant operators in terms of a subset of them called the primitive operators. These definitions can be formulated as an inductive set of rules that define the meaning  $\llbracket c \rrbracket$  of any basic constant expression  $c$ . For example, from the definition

$$e \notin S \triangleq \neg(e \in S)$$

we get the rule

$$\llbracket e \notin S \rrbracket = \neg(\llbracket e \rrbracket \in \llbracket S \rrbracket)$$

These rules define the meaning of a basic constant expression to be an expression containing only primitive constant operators and declared constants.

A basic constant expression  $e$  is a formula iff its meaning  $\llbracket e \rrbracket$  is Boolean-valued, regardless of what values are substituted for the declared constants. As explained in Section 16.1.3, this will depend on whether we are using the liberal, moderate, or conservative interpretations of the Boolean operators.

If  $S$  and  $T$  are constants declared as above, then the meaning  $\llbracket \exists x : S \subseteq T(x) \rrbracket$  of the expression  $\exists x : S \subseteq T(x)$  is the expression itself. Logicians usually carry things further, assigning some meanings  $\llbracket S \rrbracket$  and  $\llbracket T \rrbracket$  to declared constants and defining  $\llbracket \exists x : S \subseteq T(x) \rrbracket$  to equal  $\exists x : \llbracket S \rrbracket \subseteq \llbracket T \rrbracket(x)$ . For simplicity, I have short-circuited that extra level of meaning.

We are taking as given the meaning of an expression containing only primitive constant operators and declared constants. In particular, we take as primitive the notion of validity for such expressions. Section 16.1 defines the meaning of any basic constant expression in terms of these expressions, so it defines what it means for a basic constant formula to be valid.

### 16.2.2 The Meaning of a State Function

A *state* is an assignment of values to variables. (In ZF set theory, on which the semantics of TLA<sup>+</sup> is based, *value* is just another term for *set*.) States were discussed in Sections 2.1 and 2.3.

A *state function* is an expression that is built from declared variables, declared constants, and constant operators. (State functions can also contain ENABLED expressions, which are described below.) State functions are discussed on page 25 in Section 3.1. A state function assigns a constant expression to every state. If state function  $e$  assigns to state  $s$  the constant expression  $v$ , then we say that  $v$  is the value of  $e$  in state  $s$ . For example, if  $x$  is a declared variable,  $T$  is a declared constant, and  $s$  is a state that assigns to  $x$  the value 42; then the value of  $x \in T$  in state  $s$  is the constant expression  $42 \in T$ . A Boolean-valued state function is called a *state predicate*. A state predicate is *valid* iff it has the value TRUE in every state.

## Formal Semantics

A state is an assignment of values to variables. Formally, a state  $s$  is a function whose domain is the set of all variable names, where  $s[x]$  is the value that  $s$  assigns to variable  $x$ . We write  $s[x]$  instead of  $s[x]$ .

A *basic state function* is an expression that is built from declared variables, declared constants, constant operators, and ENABLED expressions, which are expressions of the form  $\text{ENABLED } e$ . An ENABLED-free basic state function is one with no ENABLED expressions. The meaning of a basic state function is a mapping from states to values. We let  $s[e]$  be the value that state function  $e$  assigns to a state  $s$ . Since a variable is a state function, we thus say both that state  $s$  assigns  $s[x]$  to variable  $x$ , and that the state function  $x$  assigns  $s[x]$  to state  $s$ .

Using the meanings assigned to the constant operators in Section 16.1 above, we inductively define  $s[e]$  for any ENABLED-free state function  $e$  to be an expression built from the primitive  $\text{TLA}^+$  constant operators, declared constants, and the values assigned by  $s$  to each variable. For example, if  $x$  is a variable and  $S$  is a constant, then

$$s[x \notin S] = \neg(s[x] \in S)$$

It is easy to see that  $s[c]$  equals  $[c]$ , for any constant expression  $c$ . (This expresses formally that a constant has the same value in all states.)

To define the meaning of all basic state function, not just ENABLED-free ones, we must define the meaning of an ENABLED expression. This is done below.

The formal semantics talks about state functions, not state predicates. Because  $\text{TLA}^+$  is typeless, there is no formal distinction between a state predicate and a state function. By a state predicate, we mean a state function  $e$  such that  $s[e]$  is Boolean-valued for every reachable state  $s$  of some specification. See the discussion of actions on pages 313–314.

I described the meaning of a state function as a “mapping” on states. This mapping cannot be a function, because there is no set of all states. Since for any set  $S$  there is a state that assigns the value  $S$  to each variable, there are too many states to form a set. (See the discussion of Russell’s paradox on page 66.) To be formal, we should define an operator  $M$  such that, if  $s$  is a state and  $e$  is a syntactically correct basic state function, then  $M(s, e)$ , which we write  $s[e]$ , is the basic constant expression that is the meaning of  $e$  in state  $s$ .

Actually, this way of describing the semantics isn’t right either. A state is a mapping from variables to values (sets), not to constant expressions. Since there are an uncountable number of sets and only a countable number of finite sequences of strings, there are values that can’t be described by any expression. Suppose  $\xi$  is such a value, and let  $s$  be a state that assigns the value  $\xi$  to the variable  $x$ . Then  $s[x = \{\}]$  equals  $\xi = \{\}$ , which isn’t a constant expression because  $\xi$  isn’t an expression. So, to be *really* formal, we would have to define a semantic constant expression to be one made from primitive constant operators,

declared constants, and arbitrary values. The meaning of a basic state function is a mapping from states to semantic constant expressions.

We won't bother with these details. Instead, we define a semi-formal semantics for basic expressions that is easier to understand. Mathematically sophisticated readers who understand the less formal exposition should be able to fill in the missing formal details.

### 16.2.3 Action Operators

A *transition function* is an expression built from state functions using the priming operator ('') and the other action operators of TLA<sup>+</sup> listed in Table 3 on page 269. A transition function assigns a value to every step, where a step is a pair of states. In a transition function, an unprimed occurrence of a variable  $x$  represents the value of  $x$  in the first (old) state, and a primed occurrence of  $x$  represents its value in the second (new) state. For example, if state  $s$  assigns the value 4 to  $x$  and state  $t$  assigns the value 5 to  $x$ , then the transition function  $x' - x$  assigns to the step  $s \rightarrow t$  the value  $5 - 4$ , which equals 1 (if  $-$  has its usual definition).

An *action* is a Boolean-valued transition function, such as  $x' > x$ . We say that action  $A$  is true on step  $s \rightarrow t$ , or that  $s \rightarrow t$  is an  $A$  step, iff  $A$  assigns the value TRUE to  $s \rightarrow t$ . An action is said to be *valid* iff it is true on any step.

The action operators of TLA<sup>+</sup> other than '' have the following meanings, where  $A$  and  $B$  are actions and  $e$  is a state function:

$$[A]_e \triangleq A \vee (e' = e)$$

$$\langle A \rangle_e \triangleq A \wedge (e' \neq e)$$

ENABLED  $A$  is the state function that is true in state  $s$  iff there is some state  $t$  such that  $s \rightarrow t$  is an  $A$  step.

$$\text{UNCHANGED } e \triangleq e' = e$$

$A \cdot B$  is the action that is true on step  $s \rightarrow t$  iff there is a state  $u$  such that  $s \rightarrow u$  is an  $A$  step and  $u \rightarrow t$  is a  $B$  step.

Priming and the construct  $[A]_v$  are introduced in Section 2.2 (page 15); the UNCHANGED operator is introduced on page 26 of Section 3.1; ENABLED is introduced on page 97 of Section 8.4; the construct  $\langle A \rangle_v$  is defined on page 91 of Section 8.1; and the action-composition operator “.” is introduced in Section 7.3 (page 76).

## Formal Semantics

A *basic transition function* is a basic expression that does not contain any temporal operators. The meaning of a basic transition function  $e$  is an assignment of a basic constant expression  $\langle s, t \rangle [e]$  to any pair of states  $\langle s, t \rangle$ . (We use here the more conventional notation  $\langle s, t \rangle$  instead of  $s \rightarrow t$ .) A transition function is valid iff  $\langle s, t \rangle [e]$  is valid, for all states  $s$  and  $t$ .

If  $e$  is a basic state function, then we interpret  $e$  as a basic transition function by defining  $\langle s, t \rangle [e]$  to equal  $s [e]$ . As indicated above, UNCHANGED and the constructs  $[A]_e$  and  $\langle A \rangle_e$  are defined in terms of priming. To define the meanings of the remaining action operators, we first define existential quantification over all states. Let  $IsAState$  be an operator such that  $IsAState(s)$  is true iff  $s$  is a state—that is, a function whose domain is the set of all variable names. (It's easy to define  $IsAState$  using the operator  $IsAFcn$ , defined on page 303.) Existential quantification over all states is then defined by

$$\exists_{\text{state}} s : p \triangleq \exists s : IsAState(s) \wedge p$$

for any formula  $p$ . The meanings of all transition functions and all state functions (including ENABLED expressions) is then defined inductively by the definitions already given and the following definitions of the remaining action operators:

$e'$  is the transition function defined by  $\langle s, t \rangle [e'] \triangleq t [e]$  for any state function  $e$ .

ENABLED  $A$  is the state function defined by

$$s [ \text{ENABLED } A ] \triangleq \exists_{\text{state}} t : \langle s, t \rangle [A]$$

for any transition function  $A$ .

$A \cdot B$  is the transition function defined by

$$\langle s, t \rangle [A \cdot B] \triangleq \exists_{\text{state}} u : \langle s, u \rangle [A] \wedge \langle u, t \rangle [B]$$

for any transition functions  $A$  and  $B$ .

The formal semantics talks about transition functions, not actions. Since TLA<sup>+</sup> is typeless, there is no formal distinction between an action and an arbitrary transition function. We could define an action  $A$  to be a transition function such that  $\langle s, t \rangle [A]$  is a Boolean for all states  $s$  and  $t$ . However, what we usually mean by an action is a transition function  $A$  such that  $\langle s, t \rangle [A]$  is a Boolean whenever  $s$  and  $t$  are reachable states of some specification. For example, a specification with a variable  $b$  of type BOOLEAN might contain an action  $b \wedge (y' = y)$ . We can calculate the meaning of ENABLED ( $b \wedge (y' = y)$ ) as follows:

$$\begin{aligned} s [ \text{ENABLED } (b \wedge (y' = y)) ] \\ = \exists_{\text{state}} t : \langle s, t \rangle [b \wedge (y' = y)] \\ = \exists_{\text{state}} t : \langle s, t \rangle [b] \wedge (\langle s, t \rangle [y'] = \langle s, t \rangle [y]) \\ = \exists_{\text{state}} t : s [b] \wedge (t [y] = s [y]) \end{aligned}$$

By definition of ENABLED.

By definition of  $\wedge$  and  $=$ .

By definition of  $'$ , since  $\langle s, t \rangle [e] = s [e]$ , for any state function  $e$ .

Types are explained on page 25.

If  $s[b]$  is a Boolean, we can now continue the calculation as follows:

$$\begin{aligned}
 \exists_{\text{state}} t : s[b] \wedge (t[y] = s[y]) \\
 = s[b] \wedge \exists_{\text{state}} t : (t[y] = s[y]) & \quad \text{By predicate logic, since } t \text{ does not occur in } s[b]. \\
 = s[b]
 \end{aligned}$$

The existence of  $t$  is obvious—for example, let it equal  $s$ .

Hence,  $s[\text{ENABLED } (b \wedge (y' = y))]$  equals  $s[b]$ , if  $s[b]$  is a Boolean. However, if  $s$  is a state that assigns the value 2 to the variable  $b$  and the value  $-7$  to the variable  $y$ , then

$$s[\text{ENABLED } (b \wedge (y' = y))] = \exists_{\text{state}} t : 2 \wedge (t[y] = -7)$$

The last expression may or may not equal 2. (See the discussion of the interpretation of the Boolean operators in Section 16.1.3 on page 296.) If the specification we are writing makes sense, it can depend on the meaning of  $\text{ENABLED } (b \wedge (y' = y))$  only for states in which the value of  $b$  is a Boolean. We don't care about its value in a state that assigns to  $b$  the value 2, just as we don't care about the value of  $3/x$  in a state that assigns the value “abc” to  $x$ . See the discussion of silly expressions in Section 6.2 (page 67).

### 16.2.4 Temporal Operators

As explained in Section 8.1, a temporal formula  $F$  is true or false for a behavior, where a behavior is a sequence of states. Syntactically, a temporal formula is defined inductively to be a state predicate or a formula having one of the forms shown in Table 4 on page 269, where  $e$  is a state function,  $A$  is an action, and  $F$  and  $G$  are temporal formulas. All the temporal operators in Table 4 are explained in Chapter 8—except for  $\Rightarrow$ , which is explained in Section 10.7 (page 156).

The formula  $\Box F$  is true for a behavior  $\sigma$  iff the temporal formula  $F$  is true for  $\sigma$  and all suffixes of  $\sigma$ . To define the constructs  $\Box[A]_e$  and  $\Diamond\langle A \rangle_e$ , we regard an action  $B$  to be a temporal formula that is true of a behavior  $\sigma$  iff the first two states of  $\sigma$  form a  $B$  step. Thus,  $\Box[A]_e$  is true of  $\sigma$  iff every successive pair of states of  $\sigma$  is a  $[A]_e$  step. All the other temporal operators of TLA<sup>+</sup>, except  $\exists$ ,  $\forall$ , and  $\Rightarrow$ , are defined as follows in terms of  $\Box$ :

$$\begin{aligned}
 \Diamond F &\triangleq \neg\Box\neg F \\
 \text{WF}_e(A) &\triangleq \Box\Diamond\neg(\text{ENABLED } \langle A \rangle_e) \vee \Box\Diamond\langle A \rangle_e \\
 \text{SF}_e(A) &\triangleq \Diamond\Box\neg(\text{ENABLED } \langle A \rangle_e) \vee \Box\Diamond\langle A \rangle_e \\
 F \rightsquigarrow G &\triangleq \Box(F \Rightarrow \Diamond G)
 \end{aligned}$$

The temporal existential quantifier  $\exists$  is a hiding operator,  $\exists x : F$  meaning formula  $F$  with the variable  $x$  hidden. To define this more precisely, we first define  $\sharp\sigma$  to be the (possibly finite) sequence of states obtained by removing

from  $\sigma$  all stuttering steps—that is, by removing any state that is the same as the previous one. We then define  $\sigma \sim_x \tau$  to be true iff  $\natural\sigma$  and  $\natural\tau$  are the same except for the values that their states assign to the variable  $x$ . Thus,  $\sigma \sim_x \tau$  is true iff  $\sigma$  can be obtained from  $\tau$  (or vice-versa) by adding and/or removing stuttering steps and changing the values assigned to  $x$  by its states. Finally,  $\exists x : F$  is defined to be true for a behavior  $\sigma$  iff  $F$  is true for some behavior  $\tau$  such that  $\sigma \sim_x \tau$ .

The temporal universal quantifier  $\forall$  is defined in terms of  $\exists$  by

$$\forall x : F \triangleq \neg(\exists x : \neg F)$$

The formula  $F \not\rightarrow G$  asserts that  $G$  does not become false before  $F$  does. More precisely, we define a formula  $H$  to be true for a finite prefix  $\rho$  of a behavior  $\sigma$  iff  $H$  is true for some (infinite) behavior that extends  $\rho$ . (In particular,  $H$  is true of the empty prefix iff  $H$  satisfies some behavior.) Then  $F \not\rightarrow G$  is defined to be true for a behavior  $\sigma$  iff (i)  $F \Rightarrow G$  is true for  $\sigma$  and (ii) for every finite prefix  $\rho$  of  $\sigma$ , if  $F$  is true for  $\rho$  then  $G$  is true for the prefix of  $\sigma$  that is one state longer than  $\rho$ .

## Formal Semantics

Formally, a behavior is a function from the set  $\text{Nat}$  of natural numbers to states. (We think of a behavior  $\sigma$  as the sequence  $\sigma[0], \sigma[1], \dots$  of states.) The meaning of a temporal formula is a predicate on behaviors—that is, a mapping from behaviors to Booleans. We write  $\sigma \models F$  for the value that the meaning of  $F$  assigns to the behavior  $\sigma$ . The temporal formula  $F$  is valid iff  $\sigma \models F$  is true, for all behaviors  $\sigma$ .

Above, we have defined all the other temporal operators in terms of  $\square$ ,  $\exists$ , and  $\not\rightarrow$ . Formally, since an action is not a temporal formula, the construct  $\square[A]_e$  is not an instance of the temporal operator  $\square$ , so its meaning should be defined separately. The construct  $\diamond(A)_e$ , which is similarly not an instance of  $\diamond$ , is then defined to equal  $\neg\square[\neg A]_e$ .

To define the meaning of  $\square$ , we first define  $\sigma^{+n}$  to be the behavior obtained by deleting the first  $n$  states of  $\sigma$ :

$$\sigma^{+n} \triangleq [i \in \text{Nat} \mapsto \sigma[i + n]]$$

We then define the meaning of  $\square$  as follows, for any temporal formula  $F$ , transition function  $A$  and state function  $e$ :

$$\begin{aligned} \sigma \models \square F &\triangleq \forall n \in \text{Nat} : \sigma^{+n} \models F \\ \sigma \models \square[A]_e &\triangleq \forall n \in \text{Nat} : \langle \sigma[n], \sigma[n + 1] \rangle \llbracket [A]_e \rrbracket \end{aligned}$$

Instead of writing  $\sigma_i$  as in Chapter 8, we use here the standard functional notation  $\sigma[i]$ .

To formalize the definition of  $\exists$  given above, we first define  $\natural$  as follows, letting  $f$  be the function such that  $\sigma[n] = \natural\sigma[f[n]]$ , for all  $n$ :

$$\begin{aligned} \natural\sigma &\triangleq \text{LET } f[n \in \text{Nat}] \triangleq \text{IF } n = 0 \text{ THEN } 0 \\ &\quad \text{ELSE IF } \sigma[n] = \sigma[n - 1] \\ &\quad \quad \text{THEN } f[n - 1] \\ &\quad \text{ELSE } f[n - 1] + 1 \\ S &\triangleq \{f[n] : n \in \text{Nat}\} \\ \text{IN } [n \in S \mapsto \sigma[\text{CHOOSE } i \in \text{Nat} : f[i] = n]] \end{aligned}$$

Next, let  $s_{x \leftarrow v}$  be the state that is the same as state  $s$  except that it assigns to the variable  $x$  the value  $v$ . We then define  $\sim_x$  by

$$\sigma \sim_x \tau \triangleq \natural\sigma = [n \in \text{DOMAIN } \natural\tau \mapsto \tau_{x \leftarrow \natural\sigma[n][x]}]$$

We next define existential quantification over behaviors. This is done much as we defined quantification over states on page 313 above; we first define *IsABehavior* so that *IsABehavior*( $\sigma$ ) is true iff  $\sigma$  is a behavior, and we then define

$$\exists_{\text{behavior}} \sigma : F \triangleq \exists \sigma : \text{IsABehavior}(\sigma) \wedge F$$

We can now define the meaning of  $\exists$  by

$$\sigma \models \exists x : F \triangleq \exists_{\text{behavior}} \tau : (\sigma \sim_x \tau) \wedge (\tau \models F)$$

Finally, we define the meaning of  $\Rightarrow$  as follows:

$$\begin{aligned} \sigma \models F \Rightarrow G &\triangleq \\ \text{LET } \text{PrefixSat}(n, H) &\triangleq \\ \exists_{\text{behavior}} \tau : \wedge \forall i \in 0 \dots (n - 1) : \tau[i] &= \sigma[i] \\ \wedge \tau \models H & \\ \text{IN } \wedge \sigma \models F \Rightarrow G & \\ \wedge \forall n \in \text{Nat} : \text{PrefixSat}(n, F) &\Rightarrow \text{PrefixSat}(n + 1, G) \end{aligned}$$

# Chapter 17

## The Meaning of a Module

Chapter 16 defines the meaning of the built-in TLA<sup>+</sup> operators. In doing so, it defines the meaning of a basic expression—that is, of an expression containing only built-in operators, declared constants, and declared variables. We now define the meaning of a module in terms of basic expressions. Since a TLA<sup>+</sup> specification consists of a collection of modules, this defines the semantics of TLA<sup>+</sup>.

We also complete the definition of the syntax of TLA<sup>+</sup> by giving the remaining context-dependent syntactic conditions not described in Chapter 15. Here's a list of some illegal expressions that satisfy the grammar of Chapter 15, and where in this chapter you can find the conditions that make them illegal.

- $F(x)$ , if  $F$  is defined by  $F(x, y) \triangleq x + y$  (Section 17.1)
- $(x' + 1)'$  (Section 17.2)
- $x + 1$ , if  $x$  is not defined or declared (Section 17.3)
- $F \triangleq 0$ , if  $F$  is already defined (Section 17.5)

This chapter is meant to be read in its entirety. To try to make it as readable as possible, I have made the exposition somewhat informal. Wherever I could, I have used examples in place of formal definitions. The examples assume that you understand the approximate meanings of the TLA<sup>+</sup> constructs, as explained in Part I. I hope that mathematically sophisticated readers will see how to fill in the missing formalism.

### 17.1 Operators and Expressions

Because it uses conventional mathematical notation, TLA<sup>+</sup> has a rather rich syntax, with several different ways of expressing the same basic type of math-

ematical operation. For example, the following expressions are all formed by applying an operator to a single argument  $e$ :

$$\text{Len}(e) \quad -e \quad \{e\} \quad e'$$

This section develops a uniform way of writing all these expressions, as well as more general kinds of expressions.

### 17.1.1 The Arity and Order of an Operator

An operator has an *arity* and an *order*. An operator's arity describes the number and order of its arguments. It's the arity of the *Len* operator that tells us that *Len*( $s$ ) is a legal expression, while *Len*( $s, t$ ) and *Len*( $+$ ) are not. All the operators of  $\text{TLA}^+$ , whether built-in or defined, fall into three classes: 0th-, 1st-, and 2nd-order operators.<sup>1</sup> Here is how these classes, and their arities, are defined:

0.  $E \triangleq x' + y$  defines  $E$  to be the 0th-order operator  $x' + y$ . A 0th-order operator takes no arguments, so it is an ordinary expression. We represent the arity of such an operator by the symbol  $\underline{\phantom{x}}$  (underscore).
1.  $F(x, y) \triangleq x \cup \{z, y\}$  defines  $F$  to be a 1st-order operator. For any expressions  $e_1$  and  $e_2$ , it defines  $F(e_1, e_2)$  to be an expression. We represent the arity of  $F$  by  $\langle \underline{\phantom{x}}, \underline{\phantom{x}} \rangle$ .

In general, a 1st-order operator takes expressions as arguments. Its arity is the tuple  $\langle \underline{\phantom{x}}, \dots, \underline{\phantom{x}} \rangle$ , with one  $\underline{\phantom{x}}$  for each argument.

2.  $G(f(\underline{\phantom{x}}, \underline{\phantom{x}}), x, y) \triangleq f(x, \{x, y\})$  defines  $G$  to be a 2nd-order operator. The operator  $G$  takes three arguments: its first argument is a 1st-order operator that takes two arguments; its last two arguments are expressions. For any operator  $Op$  of arity  $\langle \underline{\phantom{x}}, \underline{\phantom{x}} \rangle$ , and any expressions  $e_1$  and  $e_2$ , this defines  $G(Op, e_1, e_2)$  to be an expression. We say that  $G$  has arity  $\langle \langle \underline{\phantom{x}}, \underline{\phantom{x}} \rangle, \underline{\phantom{x}}, \underline{\phantom{x}} \rangle$ .

In general, the arguments of a 2nd-order operator may be expressions or 1st-order operators. A 2nd-order operator has an arity of the form  $\langle a_1, \dots, a_n \rangle$ , where each  $a_i$  is either  $\underline{\phantom{x}}$  or  $\langle \underline{\phantom{x}}, \dots, \underline{\phantom{x}} \rangle$ . (We can consider a 1st-order operator to be a degenerate case of a 2nd-order operator.)

It would be easy enough to define 3rd- and higher-order operators.  $\text{TLA}^+$  does not permit them because they are of little use and would make it harder to check level-correctness, which is discussed in Section 17.2 below.

*Len* is defined in the *Sequences* module on page 341.

---

<sup>1</sup>Even though it allows 2nd-order operators,  $\text{TLA}^+$  is still what logicians call a first-order logic because it permits quantification only over 0th-order operators. A higher-order logic would allow us to write the formula  $\exists x(\underline{\phantom{x}}) : \text{exp}$ .

## 17.1.2 $\lambda$ Expressions

When we define a 0th-order operator  $E$  by  $E \triangleq \text{exp}$ , we can write what the operator  $E$  equals—it equals the expression  $\text{exp}$ . We can explain the meaning of this definition by saying that it assigns the value  $\text{exp}$  to the symbol  $E$ . To explain the meaning of an arbitrary TLA<sup>+</sup> definition, we need to be able to write what a 1st- or 2nd-order operator equals—for example, the operator  $F$  defined by

$$F(x, y) \triangleq x \cup \{z, y\}$$

TLA<sup>+</sup> provides no way to write an expression that equals this operator  $F$ . (A TLA<sup>+</sup> expression can equal only a 0th-order operator.) We therefore generalize expressions to  $\lambda$  *expressions*, and we write the operator that  $F$  equals as the  $\lambda$  expression

$$\lambda x, y : x \cup \{z, y\}$$

The symbols  $x$  and  $y$  in this  $\lambda$  expression are called  $\lambda$  parameters. We use  $\lambda$  expressions only to explain the meaning of TLA<sup>+</sup> specifications; we can't write a  $\lambda$  expression in TLA<sup>+</sup>.

We also allow 2nd-order  $\lambda$  expressions, where the operator  $G$  defined by

$$G(f(\_, \_), x, y) \triangleq f(y, \{x, z\})$$

is equal to the  $\lambda$  expression

$$(17.1) \lambda f(\_, \_), x, y : f(y, \{x, z\})$$

The general form of a  $\lambda$  expression is  $\lambda p_1, \dots, p_n : \text{exp}$ , where  $\text{exp}$  is a  $\lambda$  expression, each parameter  $p_i$  is either an identifier  $id_i$  or has the form  $id_i(\_, \dots, \_)$ , and the  $id_i$  are all distinct. We call  $id_i$  the *identifier* of the  $\lambda$  parameter  $p_i$ . We consider the  $n = 0$  case, the  $\lambda$  expression  $\lambda : \text{exp}$  with no parameters, to be the expression  $\text{exp}$ . This makes a  $\lambda$  expression a generalization of an ordinary expression.

A  $\lambda$  parameter identifier is a bound identifier, just like the identifier  $x$  in  $\forall x : F$ . As with any bound identifiers, renaming the  $\lambda$  parameter identifiers in a  $\lambda$  expression doesn't change the meaning of the expression. For example, (17.1) is equivalent to

$$\lambda abc(\_, \_), qq, m : abc(m, \{qq, z\})$$

For obscure historical reasons, this kind of renaming is called  $\alpha$  *conversion*.

If  $Op$  is the  $\lambda$  expression  $\lambda p_1, \dots, p_n : \text{exp}$ , then  $Op(e_1, \dots, e_n)$  equals the result of replacing the identifier of the  $\lambda$  parameter  $p_i$  in  $\text{exp}$  with  $e_i$ , for all  $i$  in  $1 \dots n$ . For example,

$$(\lambda x, y : x \cup \{z, y\})(TT, w + z) = TT \cup \{z, (w + z)\}$$

This procedure for evaluating the application of a  $\lambda$  expression is called  $\beta$  *reduction*.

### 17.1.3 Simplifying Operator Application

To simplify the exposition, I assume that every operator application is written in the form  $Op(e_1, \dots, e_n)$ . TLA<sup>+</sup> provides a number of different syntactic forms for operator application, so I have to explain how they are translated into this simple form. Here are all the different forms of operator application and their translations.

- Simple constructs with a fixed number of arguments, including infix operators like  $+$ , prefix operators like `ENABLED`, and constructs like `WF`, function application, and `IF/THEN/ELSE`. These operators and constructs pose no problem. We can write  $+(a, b)$  instead of  $a + b$ ,  $IfThenElse(p, e_1, e_2)$  instead of

`IF p THEN e1 ELSE e2`

and  $Apply(f, e)$  instead of  $f[e]$ . An expression like  $a + b + c$  is an abbreviation for  $(a + b) + c$ , so it can be written  $+(+ (a, b), c)$ .

- Simple constructs with a variable number of arguments—for example,  $\{e_1, \dots, e_n\}$  and  $[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$ . We can consider each of these constructs to be repeated application of simpler operators with a fixed number of arguments. For example,

$$\begin{aligned} \{e_1, \dots, e_n\} &= \{e_1\} \cup \dots \cup \{e_n\} \\ [h_1 \mapsto e_1, \dots, h_n \mapsto e_n] &= [h_1 \mapsto e_1] @ @ \dots @ @ [h_n \mapsto e_n] \end{aligned}$$

where `@@` is defined in the *TLC* module, on page 248. Of course,  $\{e\}$  can be written *Singleton*( $e$ ) and  $[h \mapsto e]$  can be written *Record*(“ $h$ ”,  $e$ ). Note that an arbitrary CASE expression can be written in terms of CASE expressions of the form

`CASE p → e □ q → f`

using the relation

$$\begin{aligned} \text{CASE } p_1 \rightarrow e_1 \square \dots \square p_n \rightarrow e_n &= \\ \text{CASE } p_1 \rightarrow e_1 \square (p_2 \vee \dots \vee p_n) \rightarrow (\text{CASE } p_2 \rightarrow e_2 \square \dots \square p_n \rightarrow e_n) \end{aligned}$$

- Constructs that introduce bound variables—for example,

$\exists x \in S : x + z > y$

We can rewrite this expression as

`ExistsIn(S, λ x : x + z > y)`

where *ExistsIn* is a 2nd-order operator of arity  $\langle \_, \langle \_ \rangle \rangle$ . All the variants of the  $\exists$  construct can be represented as expressions using either  $\exists x \in S : e$  or  $\exists x : e$ . (Section 16.1.1 shows how these variants can be translated into expressions using only  $\exists x : e$ , but those translations don't maintain the

scoping rules—for example, rewriting  $\exists x \in S : e$  as  $\exists x : (x \in S) \wedge e$  moves  $S$  inside the scope of the bound variable  $x$ .)

All other constructs that introduce bound variables, such as  $\{x \in S : \text{exp}\}$ , can similarly be expressed in the form  $\text{Op}(e_1, \dots, e_n)$  using  $\lambda$  expressions and 2nd-order operators  $\text{Op}$ . (Chapter 16 explains how to express constructs like  $\{\langle x, y \rangle \in S : \text{exp}\}$ , which have a tuple of bound identifiers, in terms of constructs with ordinary bound identifiers.)

- Operator applications such as  $M(x)! \text{Op}(y, z)$  that arise from instantiation. We write this as  $M! \text{Op}(x, y, z)$ .
- LET expressions. The meaning of a LET expression is explained in Section 17.4 below. For now, we consider only LET-free  $\lambda$  expressions—ones that contain no LET expressions.

For uniformity, I will call an operator symbol an *identifier*, even if it is a symbol like  $+$  that isn't an identifier according to the syntax of Chapter 15.

### 17.1.4 Expressions

We can now inductively define an expression to be either a 0th-order operator, or to have the form  $\text{Op}(e_1, \dots, e_n)$  where  $\text{Op}$  is an operator and each  $e_i$  is either an expression or a 1st-order operator. The expression must be *arity-correct*, meaning that  $\text{Op}$  must have arity  $\langle a_1, \dots, a_n \rangle$ , where each  $a_i$  is the arity of  $e_i$ . In other words,  $e_i$  must be an expression if  $a_i$  equals  $\_$ ; otherwise it must be a 1st-order operator with arity  $a_i$ . We require that  $\text{Op}$  not be a  $\lambda$  expression. (If it is, we can use  $\beta$  reduction to evaluate  $\text{Op}(e_1, \dots, e_n)$  and eliminate the  $\lambda$  expression  $\text{Op}$ .) Hence, a  $\lambda$  expression can appear in an expression only as an argument of a 2nd-order operator. This implies that only 1st-order  $\lambda$  expressions can appear in an expression.

We have eliminated all bound identifiers except the ones in  $\lambda$  expressions. We maintain the TLA<sup>+</sup> requirement that an identifier that already has a meaning cannot be used as a bound identifier. Thus, in any  $\lambda$  expression  $\lambda p_1, \dots, p_n : \text{exp}$ , the identifiers of the parameters  $p_i$  cannot appear as parameter identifiers in any  $\lambda$  expression that occurs in  $\text{exp}$ .

Remember that  $\lambda$  expressions are used only to explain the semantics of TLA<sup>+</sup>. They are not part of the language, and they can't be used in a TLA<sup>+</sup> specification.

## 17.2 Levels

TLA<sup>+</sup> has a class of syntactic restrictions that come from the underlying logic TLA and have no counterpart in ordinary mathematics. The simplest of these is

that “double-priming” is prohibited. For example,  $(x' + y)'$  is not syntactically well-formed, and is therefore meaningless, because the operator ' (priming) can be applied only to a state function, not to a transition function like  $x' + y$ . This class of restriction is expressed in terms of *levels*.

In TLA, an expression has one of four basic levels, which are numbered 0, 1, 2, and 3. These levels are described below, using examples that assume  $x$ ,  $y$ , and  $c$  are declared by

VARIABLES  $x, y$       CONSTANT  $c$

and symbols like + have their usual meanings.

0. A *constant*-level expression is a constant; it contains only constants and constant operators. Example:  $c + 3$ .
1. A *state*-level expression is a state function; it may contain constants, constant operators, and unprimed variables. Example:  $x + 2 * c$ .
2. A *transition*-level expression is a transition function; it may contain anything except temporal operators. Example:  $x' + y > c$ .
3. A *temporal*-level expression is a temporal formula; it may contain any TLA operator. Example:  $\Box[x' > y + c]_{(x, y)}$ .

Chapter 16 assigns meanings to all basic expressions—ones containing only the built-in operators of TLA<sup>+</sup> and declared constants and variables. The meaning assigned to an expression depends as follows on its level.

0. The meaning of a constant-level basic expression is a constant-level basic expression containing only primitive operators.
1. The meaning of a state-level basic expression  $e$  is an assignment of a constant expression  $s[\![e]\!]$  to any state  $s$ .
2. The meaning of a transition-level basic expression  $e$  is an assignment of a constant expression  $\langle s, t \rangle[\![e]\!]$  to any transition  $s \rightarrow t$ .
3. The meaning of a temporal-level basic expression  $F$  is an assignment of a constant expression  $\sigma \models F$  to any behavior  $\sigma$ .

An expression of any level can be considered to be an expression of a higher level, except that a transition-level expression is not a temporal-level expression.<sup>2</sup> For example, if  $x$  is a declared variable, then the state-level expression  $x > 2$  is the

---

<sup>2</sup>More precisely, a transition-level expression that is not a state-level expression is not a temporal-level expression.

temporal-level formula such that  $\sigma \models x$  is the value of  $x > 2$  in the first state of  $\sigma$ , for any behavior  $\sigma$ .<sup>3</sup>

A set of simple rules inductively defines whether a basic expression is *level-correct* and, if so, what its level is. Here are some of the rules:

- A declared constant is a level-correct expression of level 0.
- A declared variable is a level-correct expression of level 1.
- If  $Op$  is declared to be a 1st-order constant operator, then the expression  $Op(e_1, \dots, e_n)$  is level-correct iff each  $e_i$  is level-correct, in which case its level is the maximum of the levels of the  $e_i$ .
- $e_1 \in e_2$  is level-correct iff  $e_1$  and  $e_2$  are, in which case its level is the maximum of the levels of  $e_1$  and  $e_2$ .
- $e'$  is level-correct and has level 2 iff  $e$  is level-correct and has level at most 1.<sup>4</sup>
- $\text{ENABLED } e$  is level-correct and has level 1 iff  $e$  is level-correct and has level at most 2.
- $\exists x : e$  is level-correct and has level  $l$  iff  $e$  is level-correct and has level  $l$ , when  $x$  is considered to be a declared constant.
- $\exists x : e$  is level-correct and has level 3 iff  $e$  is level-correct and has any level other than 2, when  $x$  is considered to be a declared variable.

There are similar rules for the other TLA<sup>+</sup> operators.

A useful consequence of these rules is that level-correctness of a basic expression does not depend on the levels of the declared identifiers. In other words, an expression  $e$  is level-correct when  $c$  is declared to be a constant iff it is level-correct when  $c$  is declared to be a variable. Of course, the level of  $e$  may depend on the level of  $c$ .

We can abstract these rules by generalizing the concept of a level. So far, we have defined the level only of an expression. We can define the level of a 1st- or 2nd-order operator  $Op$  to be a rule for determining the level-correctness and level of an expression  $Op(e_1, \dots, e_n)$  as a function of the levels of the arguments  $e_i$ . The level of a 1st-order operator is a rule, so the level of a 2nd-order operator  $Op$  is a rule that depends in part on rules—namely, on the levels of the arguments that are operators. This makes a rigorous general definition of levels for 2nd-order operators rather complicated. Fortunately, there's a simpler, less general

---

<sup>3</sup>The expression  $x + 2$  can be considered to be a temporal-level expression that, like the temporal-level expression  $\Box(x + 2)$ , is silly. (See the discussion of silliness in Section 6.2 on page 67.)

<sup>4</sup>If  $e$  is a constant expression, then  $e'$  equals  $e$ , so we could consider  $e'$  to have level 0. For simplicity, we consider  $e'$  to have level 2 even if  $e$  is a constant.

definition that handles all the operators of TLA<sup>+</sup>. Even more fortunately, you don't have to know it, so I won't bother writing it down. All you need to know is that there exists a way of assigning a level to every built-in operator of TLA<sup>+</sup>. The level-correctness and level of any basic expression is then determined by those levels and the levels of the declared identifiers that occur in the expression.

One important class of operator levels are the *constant* levels. Any expression built from constant-level operators and declared constants has constant level. The built-in constant operators of TLA<sup>+</sup>, listed in Tables 1 and 2 (pages 268 and 269) all have constant level. Any operator defined solely in terms of constant-level operators and declared constants has constant level.

We now extend the definition of level-correctness from expressions to  $\lambda$  expressions. We define the  $\lambda$  expression  $\lambda p_1, \dots, p_n : \text{exp}$  to be level-correct iff  $\text{exp}$  is level-correct when the  $\lambda$  parameter identifiers are declared to be constants of the appropriate arity. For example,  $\lambda p, q(\_) : \text{exp}$  is level-correct iff  $\text{exp}$  is level-correct with the additional declaration

CONSTANTS  $p, q(\_)$

This inductively defines level-correctness for  $\lambda$  expressions. The definition is reasonable because, as observed a few paragraphs ago, the level-correctness of  $\text{exp}$  doesn't depend on whether we assign level 0 or 1 to the  $\lambda$  parameters. One can also define the level of an arbitrary  $\lambda$  expression, but that would require the general definition of the level of an operator, which we want to avoid.

## 17.3 Contexts

Syntactic correctness of a basic expression depends on the arities of the declared identifiers. The expression  $\text{Foo} = \{\}$  is syntactically correct if  $\text{Foo}$  is declared to be a variable, and hence of arity  $\_$ , but not if it's declared to be a (1st-order) constant of arity  $\langle \_ \rangle$ . The meaning of a basic expression also depends on the levels of the declared identifiers. We can't determine those arities and levels just by looking at the expression itself; they are implied by the context in which the expression appears. A nonbasic expression contains defined as well as declared operators. Its syntactic correctness and meaning depend on the definitions of those operators, which also depend on the context. This section defines a precise notion of a context.

For uniformity, built-in operators are treated the same as defined and declared operators. Just as the context might tell us that the identifier  $x$  is a declared variable, it tells us that  $\in$  is declared to be a constant-level operator of arity  $\langle \_, \_ \rangle$  and that  $\notin$  is defined to equal  $\lambda a, b : \neg(\in(a, b))$ . We assume a standard context that specifies all the built-in operators of TLA<sup>+</sup>.

To define contexts, let's first define declarations and definitions. A *declaration* assigns an arity and level to an operator name. A *definition* assigns a LET-free  $\lambda$  expression to an operator name. A *module definition* assigns the meaning

of a module to a module name, where the meaning of a module is defined in Section 17.5 below.<sup>5</sup> A *context* consists of a set of declarations, definitions, and module definitions such that

- C1. An operator name is declared or defined at most once by the context.  
(This means that it can't be both declared and defined.)
- C2. No operator defined or declared by the context appears as the identifier of a  $\lambda$  parameter in any definition's expression.
- C3. Every operator name that appears in a definition's expression is either a  $\lambda$  parameter's identifier or is declared (not defined) by the context.
- C4. No module name is assigned meanings by two different module definitions.

Module and operator names are handled separately. The same string may be both a module name that is defined by a module definition and an operator name that is either declared or defined by an ordinary definition.

Here is an example of a context that declares the symbols  $\cup$ ,  $a$ ,  $b$ , and  $\in$ , defines the symbols  $c$  and *foo*, and defines the module *Naturals*:

$$(17.2) \{ \cup : \langle \_, \_ \rangle, \quad a : \_, \quad b : \_, \quad \in : \langle \_, \_ \rangle, \quad c \triangleq \cup(a, b), \\ \text{foo} \triangleq \lambda p, q(\_) : \in(p, \cup(q(b), a)), \quad \text{Naturals} \stackrel{m}{=} \dots \}$$

Not shown are the levels assigned to the operators  $\cup$ ,  $a$ ,  $b$ , and  $\in$  and the meaning assigned to *Naturals*.

If  $\mathcal{C}$  is a context, a  $\mathcal{C}$ -basic  $\lambda$  expression is defined to be a  $\lambda$  expression that contains only symbols declared in  $\mathcal{C}$  (in addition to  $\lambda$  parameters). For example,  $\lambda x : \in(x, \cup(a, b))$  is a  $\mathcal{C}$ -basic  $\lambda$  expression if  $\mathcal{C}$  is the context (17.2). However, neither  $\cap(a, b)$  nor  $\lambda x : c(x, b)$  is a  $\mathcal{C}$ -basic  $\lambda$  expression because neither  $\cap$  nor  $c$  is declared in  $\mathcal{C}$ . (The symbol  $c$  is defined, not declared, in  $\mathcal{C}$ .) A  $\mathcal{C}$ -basic  $\lambda$  expression is *syntactically correct* if it is arity- and level-correct with the arities and levels assigned by  $\mathcal{C}$  to the expression's operators. Condition C3 states that if  $Op \triangleq exp$  is a definition in context  $\mathcal{C}$ , then  $exp$  is a  $\mathcal{C}$ -basic  $\lambda$  expression. We add to C3 the requirement that it be syntactically correct.

We also allow a context to contain a special definition of the form  $Op \triangleq ?$  that assigns to the name  $Op$  an “illegal” value  $?$  that is not a  $\lambda$  expression. This definition indicates that, in the context, it is illegal to use the operator name  $Op$ .

## 17.4 The Meaning of a $\lambda$ Expression

We now define the meaning  $\mathcal{C}[e]$  of a  $\lambda$  expression  $e$  in a context  $\mathcal{C}$  to be a  $\mathcal{C}$ -basic  $\lambda$  expression. If  $e$  is an ordinary (nonbasic) expression, and  $\mathcal{C}$  is the

---

<sup>5</sup>The meaning of a module is defined in terms of contexts, so these definitions appear to be circular. In fact, the definitions of context and of the meaning of a module together form a single inductive definition.

context that specifies the built-in TLA<sup>+</sup> operators and declares the constants and variables that occur in  $e$ , then this will define  $\mathcal{C}[e]$  to be a basic expression. Since Chapter 16 defines the meaning of basic expressions, this defines the meaning of an arbitrary expression. The expression  $e$  may contain LET constructs, so this defines the meaning of LET, the one operator whose meaning is not defined in Chapter 16.

Basically,  $\mathcal{C}[e]$  is obtained from  $e$  by replacing all defined operator names with their definitions and then applying  $\beta$  reduction whenever possible. Recall that  $\beta$  reduction replaces

$$(\lambda p_1, \dots, p_n : \text{exp}) (e_1, \dots, e_n)$$

with the expression obtained from  $\text{exp}$  by replacing the identifier of  $p_i$  with  $e_i$ , for each  $i$ . The definition of  $\mathcal{C}[e]$  does not depend on the levels assigned by the declarations of  $\mathcal{C}$ . So, we ignore levels in the definition. The inductive definition of  $\mathcal{C}[e]$  consists of the following rules:

- If  $e$  is an operator symbol, then  $\mathcal{C}[e]$  equals (i)  $e$  if  $e$  is declared in  $\mathcal{C}$ , or (ii) the  $\lambda$  expression of  $e$ 's definition in  $\mathcal{C}$  if  $e$  is defined in  $\mathcal{C}$ .
- If  $e$  is  $Op(e_1, \dots, e_n)$ , where  $Op$  is declared in  $\mathcal{C}$ , then  $\mathcal{C}[e]$  equals the expression  $Op(\mathcal{C}[e_1], \dots, \mathcal{C}[e_n])$ .
- If  $e$  is  $Op(e_1, \dots, e_n)$ , where  $Op$  is defined in  $\mathcal{C}$  to equal the  $\lambda$  expression  $d$ , then  $\mathcal{C}[e]$  equals the  $\beta$  reduction of  $\overline{d}(\mathcal{C}[e_1], \dots, \mathcal{C}[e_n])$ , where  $\overline{d}$  is obtained from  $d$  by  $\alpha$  conversion (replacement of  $\lambda$  parameters) so that no  $\lambda$  parameter's identifier appears in both  $\overline{d}$  and some  $\mathcal{C}[e_i]$ .
- If  $e$  is  $\lambda p_1, \dots, p_n : \text{exp}$ , then  $\mathcal{C}[e]$  equals  $\lambda p_1, \dots, p_n : \mathcal{D}[\text{exp}]$ , where  $\mathcal{D}$  is the context obtained by adding to  $\mathcal{C}$  the declarations that, for each  $i$  in  $1 \dots n$ , assign to the  $i^{\text{th}}$   $\lambda$  parameter's identifier the arity determined by  $p_i$ .
- If  $e$  is where  $d$  is a  $\lambda$  expression and  $\text{exp}$  an expression, then  $\mathcal{C}[e]$  equals  $\mathcal{D}[\text{exp}]$ , where  $\mathcal{D}$  is the context obtained by adding to  $\mathcal{C}$  the definition that assigns  $\mathcal{C}[d]$  to  $Op$ .
- If  $e$  is

$$\text{LET } Op(p_1, \dots, p_n) \triangleq \text{INSTANCE } \dots \text{ IN } \text{exp}$$

then  $\mathcal{C}[e]$  equals  $\mathcal{D}[\text{exp}]$ , where  $\mathcal{D}$  is the new current context obtained by “evaluating” the statement

$$Op(p_1, \dots, p_n) \triangleq \text{INSTANCE } \dots$$

in the current context  $\mathcal{C}$ , as described in Section 17.5.5 below.

The last two conditions define the meaning of any LET construct, because

- The operator definition  $Op(p_1, \dots, p_n) \triangleq d$  in a LET means

$$Op \triangleq \lambda p_1, \dots, p_n : d$$

- A function definition  $Op[x \in S] \triangleq d$  in a LET means

$$Op \triangleq \text{CHOOSE } Op : Op = [x \in S \mapsto d]$$

- The expression LET  $Op_1 \triangleq d_1 \dots Op_n \triangleq d_n$  IN  $exp$  is defined to equal

$$\text{LET } Op_1 \triangleq d_1 \text{ IN } (\text{LET } \dots \text{ IN } (\text{LET } Op_n \triangleq d_n \text{ IN } exp) \dots)$$

The  $\lambda$  expression  $e$  is defined to be legal (syntactically well-formed) in the context  $\mathcal{C}$  iff these rules define  $\mathcal{C}[e]$  to be a legal  $\mathcal{C}$ -basic expression.

## 17.5 The Meaning of a Module

The meaning of a module depends on a context. For an external module, which is not a submodule of another module, the context consists of declarations and definitions of all the built-in operators of TLA<sup>+</sup>, together with definitions of some other modules. Section 17.7 below discusses where the definitions of those other modules come from.

The meaning of a module in a context  $\mathcal{C}$  consists of six sets:

*Dcl* A set of declarations. They come from CONSTANT and VARIABLE declarations and declarations in extended modules (modules appearing in an EXTENDS statement).

*GDef* A set of global definitions. They come from ordinary (non-LOCAL) definitions and global definitions in extended and instantiated modules.

*LDef* A set of local definitions. They come from LOCAL definitions and LOCAL instantiations of modules. (Local definitions are not obtained by other modules that extend or instantiate the module.)

*MDef* A set of module definitions. They come from submodules of the module and of extended modules.

*Ass* A set of assumptions. They come from ASSUME statements and from extended modules.

*Thm* A set of theorems. They come from THEOREM statements, from theorems in extended modules, and from the assumptions and theorems of instantiated modules, as explained in Section 17.5.5 below.

The  $\lambda$  expressions of definitions in  $GDef$  and  $LDef$ , as well as the expressions in  $Ass$  and  $Thm$ , are  $(\mathcal{C} \cup Dcl)$ -basic  $\lambda$  expressions. In other words, the only operator symbols they contain (other than  $\lambda$  parameter identifiers) are ones declared in  $\mathcal{C}$  or in  $Dcl$ .

The meaning of a module in a context  $\mathcal{C}$  is defined by an algorithm for computing these six sets. The algorithm processes each statement in the module in turn, from beginning to end. The meaning of the module is the value of those sets when the end of the module is reached.

Initially, all six sets are empty. The rules for handling each possible type of statement are given below. In these rules, the *current context*  $\mathcal{CC}$  is defined to be the union of  $\mathcal{C}$ ,  $Dcl$ ,  $GDef$ ,  $LDef$ , and  $MDef$ .

When the algorithm adds elements to the context  $\mathcal{CC}$ , it uses  $\alpha$  conversion to ensure that no defined or declared operator name appears as a  $\lambda$  parameter's identifier in any  $\lambda$  expression in  $\mathcal{CC}$ . For example, if the definition  $foo \triangleq \lambda x : x + 1$  is in  $LDef$ , then adding a declaration of  $x$  to  $Dcl$  requires  $\alpha$  conversion of this definition to rename the  $\lambda$  parameter identifier  $x$ . This  $\alpha$  conversion is not explicitly mentioned.

### 17.5.1 Extends

An EXTENDS statement has the form

EXTENDS  $M_1, \dots, M_n$

where each  $M_i$  is a module name. This statement must be the first one in the module. The statement sets the values of  $Dcl$ ,  $GDef$ ,  $MDef$ ,  $Ass$ , and  $Thm$  equal to the union of the corresponding values for the module meanings assigned by  $\mathcal{C}$  to the module names  $M_i$ .

This statement is legal iff the module names  $M_i$  are all defined in  $\mathcal{C}$ , and the resulting current context  $\mathcal{CC}$  does not assign more than one meaning to any symbol. More precisely, if the same symbol is defined or declared by two or more of the  $M_i$ , then those duplicate definitions or declarations must all have been obtained through a (possibly empty) chains of EXTENDS statements from the same definition or declaration. For example, suppose  $M_1$  extends the *Naturals* module, and  $M_2$  extends  $M_1$ . Then the three modules *Naturals*,  $M_1$ , and  $M_2$  all define the operator  $+$ . The statement

EXTENDS *Naturals*,  $M_1, M_2$

can still be legal, because each of the three definitions is obtained by a chain of EXTENDS statements (of length 0, 1, and 2, respectively) from the definition of  $+$  in the *Naturals* module.

When decomposing a large specification into modules, we often want a module  $M$  to extend modules  $M_1, \dots, M_n$ , where the  $M_i$  have declared constants

and/or variables in common. In this case, we put the common declarations in a module  $P$  that is extended by all the  $M_i$ .

## 17.5.2 Declarations

A declaration statement has one of the forms

$$\text{CONSTANT } c_1, \dots, c_n \quad \text{VARIABLE } v_1, \dots, v_n$$

where each  $v_i$  is an identifier and each  $c_i$  is either an identifier or has the form  $Op(-, \dots, -)$ , for some identifier  $Op$ . This statement adds to the set  $Dcl$  the obvious declarations. It is legal iff none of the declared identifiers is defined or declared in  $\mathcal{CC}$ .

## 17.5.3 Operator Definitions

A global operator definition<sup>6</sup> has one of the two forms

$$Op \triangleq exp \quad Op(p_1, \dots, p_n) \triangleq exp$$

where  $Op$  is an identifier,  $exp$  is an expression, and each  $p_i$  is either an identifier or has the form  $P(-, \dots, -)$ , where  $P$  is an identifier. We consider the first form an instance of the second with  $n = 0$ .

This statement is legal iff  $Op$  is not declared or defined in  $\mathcal{CC}$  and the  $\lambda$  expression  $\lambda p_1, \dots, p_n : exp$  is legal in context  $\mathcal{CC}$ . In particular, no  $\lambda$  parameter in this  $\lambda$  expression can be defined or declared in  $\mathcal{CC}$ . The statement adds to  $GDef$  the definition that assigns to  $Op$  the  $\lambda$  expression  $\mathcal{CC}[\lambda p_1, \dots, p_n : exp]$ .

A local operator definition has one of the two forms

$$\text{LOCAL } Op \triangleq exp \quad \text{LOCAL } Op(p_1, \dots, p_n) \triangleq exp$$

It is the same as a global definition, except that it adds the definition to  $LDef$  instead of  $GDef$ .

## 17.5.4 Function Definitions

A global function definition has the form

$$Op[fcnargs] \triangleq exp$$

---

<sup>6</sup>An operator definition statement should not be confused with a definition clause in a LET expression. The meaning of a LET expression is described in Section 17.4.

where  $fcnargs$  is a comma-separated list of elements, each having the form  $Id_1, \dots, Id_n \in S$  or  $\langle Id_1, \dots, Id_n \rangle \in S$ . It is equivalent to the global operator definition

$$Op \triangleq \text{CHOOSE } Op : Op = [fcnargs \mapsto exp]$$

A local function definition, which has the form

$$\text{LOCAL } Op[fcnargs] \triangleq exp$$

is equivalent to the analogous local operator definition.

### 17.5.5 Instantiation

We consider first a global instantiation of the form

$$(17.3) \quad I(p_1, \dots, p_m) \triangleq \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \dots, q_n \leftarrow e_n$$

For this to be legal,  $N$  must be a module name defined in  $\mathcal{CC}$ . Let  $NDcl$ ,  $NDef$ ,  $NAss$ , and  $NThm$  be the sets  $Dcl$ ,  $GDef$ ,  $Ass$ , and  $Thm$  in the meaning assigned to  $N$  by  $\mathcal{CC}$ . The  $q_i$  must be distinct identifiers declared by  $NDcl$ . We add a WITH clause of the form  $Op \leftarrow Op$  for any identifier  $Op$  that is declared in  $NDcl$  but is not one of the  $q_i$ , so the  $q_i$  constitute all the identifiers declared in  $NDcl$ .

Neither  $I$  nor any of the identifiers of the definition parameters  $p_i$  may be defined or declared in  $\mathcal{CC}$ . Let  $\mathcal{D}$  be the context obtained by adding to  $\mathcal{CC}$  the obvious constant-level declaration for each  $p_i$ . Then  $e_i$  must be syntactically well-formed in the context  $\mathcal{D}$ , and  $\mathcal{D}[\![e_i]\!]$  must have the same arity as  $q_i$ , for each  $i \in 1 \dots n$ .

The instantiation must also satisfy the following level-correctness condition. Define module  $N$  to be a *constant* module iff every declaration in  $NDcl$  has constant level, and every operator appearing in every definition in  $NDef$  has constant level. If  $N$  is *not* a constant module, then for each  $i$  in  $1 \dots n$ :

- If  $q_i$  is declared in  $NDcl$  to be a constant operator, then  $\mathcal{D}[\![e_i]\!]$  has constant level.
- If  $q_i$  is declared in  $NDcl$  to be a variable (a 0th-order operator of level 1), then  $\mathcal{D}[\![e_i]\!]$  has level 0 or 1.

The reason for this condition is explained in Section 17.8 below.

For each definition  $Op \triangleq \lambda r_1, \dots, r_p : e$  in  $NDef$ , the definition

$$(17.4) \quad I!Op \triangleq \lambda p_1, \dots, p_m, r_1, \dots, r_p : \overline{e}$$

is added to  $GDef$ , where  $\overline{e}$  is the expression obtained from  $e$  by substituting  $e_i$  for  $q_i$ , for all  $i \in 1 \dots n$ . Before doing this substitution,  $\alpha$  conversion must be

applied to ensure that  $\mathcal{CC}$  is a correct context after the definition of  $I!Op$  is added to  $GDef$ . The precise definition of  $\overline{e}$  is a bit subtle; it is given in Section 17.8 below. We require that the  $\lambda$  expression in (17.4) be level-correct. (If  $N$  is a nonconstant module, then level-correctness of this  $\lambda$  expression is implied by the level condition on parameter instantiation described in the preceding paragraph.) Legality of the definition of  $Op$  in module  $N$  and of the WITH substitutions implies that the  $\lambda$  expression is arity-correct in the current context. Remember that  $I!Op(c_1, \dots, c_m, d_1, \dots, d_n)$  is actually written in TLA<sup>+</sup> as  $I(c_1, \dots, c_m)!Op(d_1, \dots, d_n)$ .

Also added to  $GDef$  is the special definition  $I \triangleq ?$ . This prevents  $I$  from later being defined or declared as an operator name.

If  $NAss$  equals the set  $\{A_1, \dots, A_k\}$  of assumptions, then for each theorem  $T$  in  $NThm$ , we add to  $Thm$  the theorem

$$\overline{A_1} \wedge \dots \wedge \overline{A_k} \Rightarrow \overline{T}$$

(As above,  $\overline{T}$  and the  $\overline{A_j}$  are obtained from  $T$  and the  $A_j$  by substituting  $e_i$  for  $q_i$ , for each  $i$  in  $1 \dots k$ .)

A global INSTANCE statement can also have the two forms

$$\begin{aligned} I &\triangleq \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \dots, q_n \leftarrow e_n \\ &\text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \dots, q_n \leftarrow e_n \end{aligned}$$

The first is just the  $m = 0$  case of (17.3); the second is similar to the first, except the definitions added to  $GDef$  do not have  $I!$  prepended to the operator names. The second form also has the legality condition that none of the defined symbols in  $N$  may be defined or declared in the current context, except in the following case. An operator definition may be included multiple times through chains of INSTANCE and EXTENDS statements if it is defined in a module<sup>7</sup> having no declarations. For example, suppose the current context contains a definition of  $+$  obtained through extending the *Naturals* module. Then an INSTANCE  $N$  statement is legal even though  $N$  also extends *Naturals* and therefore defines  $+$ . Because the *Naturals* module declares no parameters, instantiation cannot change the definition of  $+$ .

In all forms of the INSTANCE statement, omitting the WITH clause is equivalent to the case  $n = 0$  of these statements. (Remember that all the declared identifiers of module  $N$  are either explicitly or implicitly instantiated.)

A local INSTANCE statement consists of the keyword LOCAL followed by an INSTANCE statement of the form described above. It is handled in a similar fashion to a global INSTANCE statement, except that all definitions are added to  $LDef$  instead of  $GDef$ .

---

<sup>7</sup>An operator  $J!Op$  is defined in the module that contains the  $J \triangleq \text{INSTANCE} \dots$  statement.

### 17.5.6 Theorems and Assumptions

A theorem has one of the forms

THEOREM  $exp$       THEOREM  $Op \triangleq exp$

where  $exp$  is an expression, which must be legal in the current context  $\mathcal{CC}$ . The first form adds the theorem  $\mathcal{CC}[\![exp]\!]$  to the set  $Thm$ . The second form is equivalent to the two statements

$Op \triangleq exp$   
THEOREM  $Op$

An assumption has one of the forms

ASSUME  $exp$       ASSUME  $Op \triangleq exp$

The expression  $exp$  must have constant level. An assumption is similar to a theorem except that  $\mathcal{CC}[\![exp]\!]$  is added to the set  $Ass$ .

### 17.5.7 Submodules

A module can contain a submodule, which is a complete module that begins with

————— MODULE  $N$  —————

for some module name  $N$ , and ends with

—————

This is legal iff the module name  $N$  is not defined in  $\mathcal{CC}$  and the module is legal in the context  $\mathcal{CC}$ . In this case, the module definition that assigns to  $N$  the meaning of the submodule in context  $\mathcal{CC}$  is added to  $MDef$ .

A submodule can be used in an INSTANCE statement that appears either later in the current module or in a module that extends the current module. Submodules of a module  $M$  are *not* added to the set  $MDef$  of a module that instantiates  $M$ .

## 17.6 Correctness of a Module

Section 17.5 above defines the meaning of a module to consist of the six sets  $Dcl$ ,  $GDef$ ,  $LDef$ ,  $MDef$ ,  $Ass$ , and  $Thm$ . Mathematically, we can view the meaning of a module to be the assertion that all the theorems in  $Thm$  are consequences

of the assumptions in  $\text{Ass}$ . More precisely, let  $A$  be the conjunction of all the assumptions in  $\text{Ass}$ . The module asserts that, for every theorem  $T$  in  $\text{Thm}$ , the formula  $A \Rightarrow T$  is valid.<sup>8</sup>

An assumption or theorem of the module is a  $(\mathcal{C} \cup \text{Dcl})$ -basic expression. For an outermost module (not a submodule),  $\mathcal{C}$  declares only the built-in operators of  $\text{TLA}^+$ , and  $\text{Dcl}$  declares the declared constants and variables of the module. Therefore, each formula  $A \Rightarrow T$  asserted by the module is a basic expression. We say that the module is *semantically correct* if each of these expressions  $A \Rightarrow T$  is a valid formula in the context  $\text{Dcl}$ . Chapter 16 defines what it means for a basic expression to be a valid formula.

By defining the meaning of a theorem, we have defined the meaning of a  $\text{TLA}^+$  specification. Any mathematically meaningful question we can ask about a specification can be framed as the question of whether a certain formula is a valid theorem.

## 17.7 Finding Modules

For a module  $M$  to have a meaning in a context  $\mathcal{C}$ , every module  $N$  extended or instantiated by  $M$  must have its meaning defined in  $\mathcal{C}$ —unless  $N$  is a submodule of  $M$  or of a module extended by  $M$ . In principle, module  $M$  is interpreted in a context containing declarations and definitions of the built-in  $\text{TLA}^+$  operator names and module definitions of all modules needed to interpret  $M$ . In practice, a tool (or a person) begins interpreting  $M$  in a context  $\mathcal{C}_0$  initially containing only declarations and definitions of the built-in  $\text{TLA}^+$  operator names. When the tool encounters an EXTENDS or INSTANCE statement that mentions a module named  $N$  not defined in the current context  $\mathcal{CC}$  of  $M$ , the tool finds the module named  $N$ , interprets it in the context  $\mathcal{C}_0$ , and then adds the module definition for  $N$  to  $\mathcal{C}_0$  and to  $\mathcal{CC}$ .

The definition of the  $\text{TLA}^+$  language does not specify how a tool finds a module named  $N$ . A tool will most likely look for the module in a file named  $N.tla$ .

The meaning of a module depends on the meanings of the modules that it extends or instantiates. The meaning of each of those modules in turn may depend on the meanings of other modules, and so on. Thus, the meaning of a module depends on the meanings of some set of modules. A module  $M$  is syntactically incorrect if this set of modules includes  $M$  itself.

---

<sup>8</sup>In a temporal logic like  $\text{TLA}$ , the formula  $F \Rightarrow G$  is not in general equivalent to the assertion that  $G$  is a consequence of assumption  $F$ . However, the two are equivalent if  $F$  is a constant formula, and  $\text{TLA}^+$  allows only constant assumptions.

## 17.8 The Semantics of Instantiation

Section 17.5.5 above defines the meaning of an INSTANCE statement in terms of substitution. I now define precisely how that substitution is performed and explain the level-correctness rule for instantiating nonconstant modules.

Suppose that module  $M$  contains the statement

$$I \triangleq \text{INSTANCE } N \text{ WITH } q_1 \leftarrow e_1, \dots, q_n \leftarrow e_n$$

where the  $q_i$  are all the declared identifiers of module  $N$ , and that  $N$  contains the definition

$$F \triangleq e$$

where no  $\lambda$  parameter identifier in  $e$  is defined or declared in the current context of  $M$ . The INSTANCE statement then adds to the current context of  $M$  the definition

$$(17.5) \quad I!F \triangleq \bar{e}$$

where  $\bar{e}$  is obtained from  $e$  by substituting  $e_i$  for  $q_i$ , for all  $i$  in  $1 \dots n$ .

A fundamental principle of mathematics is that substitution preserves validity; substituting in a valid formula yields a valid formula. So, we want to define  $\bar{e}$  so that, if  $F$  is a valid formula in  $N$ , then  $I!F$  is a valid formula in  $M$ .

A simple example shows that the level rule for instantiating nonconstant modules is necessary to preserve the validity of  $F$ . Suppose  $F$  is defined to equal  $\square[c' = c]_c$ , where  $c$  is declared in  $N$  to be a constant. Then  $F$  is a temporal formula asserting that no step changes  $c$ . It is valid because a constant has the same value in every state of a behavior. If we allowed an instantiation that substitutes a variable  $x$  for the constant  $c$ , then  $I!F$  would be the formula  $\square[x' = x]_x$ . This is not a valid formula because it is false for any behavior in which the value of  $x$  changes. Since  $x$  is a variable, such a behavior obviously exists. Preserving validity requires that we not allow substitution of a nonconstant for a declared constant when instantiating a nonconstant module. (Since  $\square$  and  $'$  are nonconstant operators, this definition of  $F$  can appear only in a nonconstant module.)

In ordinary mathematics, there is one tricky problem in making substitution preserve validity. Consider the formula

$$(17.6) \quad (n \in \text{Nat}) \Rightarrow (\exists m \in \text{Nat} : m \geq n)$$

This formula is valid because it is true for any value of  $n$ . Now, suppose we substitute  $m + 1$  for  $n$ . A naive substitution that simply replaces  $n$  by  $m + 1$  would yield the formula

$$(17.7) \quad (m + 1 \in \text{Nat}) \Rightarrow (\exists m \in \text{Nat} : m \geq m + 1)$$

Since the formula  $\exists m \in \text{Nat} : m \geq m + 1$  is equivalent to FALSE, (17.7) is obviously not valid. Mathematicians call this problem *variable capture*;  $m$  is “captured” by the quantifier  $\exists m$ . Mathematicians avoid it by the rule that, when substituting for an identifier in a formula, one does not substitute for bound occurrences of the identifier. This rule requires that  $m$  be removed from (17.6) by  $\alpha$  conversion before  $m + 1$  is substituted for  $n$ .

Section 17.5.5 defines the meaning of the INSTANCE statement in a way that avoids variable capture. Indeed, formula (17.7) is illegal in TLA<sup>+</sup> because the subexpression  $m + 1 \in \text{Nat}$  is allowed only in a context in which  $m$  is defined or declared, in which case  $m$  cannot be used as a bound identifier, so the subexpression  $\exists m \dots$  is illegal. The  $\alpha$  conversion necessary to produce a syntactically well-formed expression makes this kind of variable capture impossible.

The problem of variable capture occurs in a more subtle form in certain nonconstant operators of TLA<sup>+</sup>, where it is not prevented by the syntactic rules. Most notable of these operators is ENABLED. Suppose  $x$  and  $y$  are declared variables of module  $N$ , and  $F$  is defined by

$$F \triangleq \text{ENABLED } (x' = 0 \wedge y' = 1)$$

Then  $F$  is equivalent to TRUE, so it is valid in module  $N$ . (For any state  $s$ , there exists a state  $t$  in which  $x = 0$  and  $y = 1$ .) Now suppose  $z$  is a declared variable of module  $M$ , and let the instantiation be

$$I \triangleq \text{INSTANCE } N \text{ WITH } x \leftarrow z, y \leftarrow z$$

With naive substitution,  $I!F$  would equal

$$\text{ENABLED } (z' = 0 \wedge z' = 1)$$

which is equivalent to FALSE. (For any state  $s$ , there is no state  $t$  in which  $z = 0$  and  $z = 1$  are both true.) Hence,  $I!F$  would not be a theorem, so instantiation would not preserve validity.

Naive substitution in a formula of the form ENABLED  $A$  does not preserve validity because the primed variables in  $A$  are really bound identifiers. The formula ENABLED  $A$  asserts that *there exist* values of the primed variables such that  $A$  is true. Substituting  $z'$  for  $x'$  and  $y'$  in the ENABLED formula is really substitution for a bound identifier. It isn't ruled out by the syntactic rules of TLA<sup>+</sup> because the quantification is implicit.

To preserve validity, we must define  $\overline{e}$  in (17.5) so it avoids capture of identifiers implicitly bound in ENABLED expressions. Before performing the substitution, we first replace the primed occurrences of variables in ENABLED expressions with new variable symbols. That is, for each subexpression of  $e$  of the form ENABLED  $A$  and each declared variable  $q$  of module  $N$ , we replace every primed occurrence of  $q$  in  $A$  with a new symbol, which we write  $\$q$ , that does not appear in  $A$ . This new symbol is considered to be bound by the ENABLED operator. For example, the module

| MODULE $N$           |                                           |
|----------------------|-------------------------------------------|
| VARIABLE $u$         |                                           |
| $G(v, A) \triangleq$ | ENABLED $(A \vee (\{u, v\}' = \{u, v\}))$ |
| $H \triangleq$       | $(u' = u) \wedge G(u, u' \neq u)$         |

has as its global definitions the set

$$\{ G \triangleq \lambda v, A : \text{ENABLED } (A \vee (\{u, v\}' = \{u, v\})), \\ H \triangleq (u' = u) \wedge \text{ENABLED } ((u' \neq u) \vee (\{u, u\}' = \{u, u\})) \}$$

The statement

$$I \triangleq \text{INSTANCE } N \text{ WITH } u \leftarrow x$$

adds the following definitions to the current module:

$$I!G \triangleq \lambda v, A : \text{ENABLED } (A \vee (\{\$u, v\}' = \{x, v\})) \\ I!H \triangleq (x' = x) \wedge \text{ENABLED } ((\$u' \neq x) \vee (\{\$u, \$u\}' = \{x, x\}))$$

Observe that  $I!H$  does not equal  $(x' = x) \wedge I!G(x, x' \neq x)$ , even though  $H$  equals  $(u' = u) \wedge G(u, u' \neq u)$  in module  $N$  and the instantiation substitutes  $x$  for  $u$ .

As another example, consider the module

| MODULE $N$        |                               |
|-------------------|-------------------------------|
| VARIABLES $u, v$  |                               |
| $A \triangleq$    | $(u' = u) \wedge (v' \neq v)$ |
| $B(d) \triangleq$ | ENABLED $d$                   |
| $C \triangleq$    | $B(A)$                        |

The instantiation

$$I \triangleq \text{INSTANCE } N \text{ WITH } u \leftarrow x, v \leftarrow x$$

adds the following definitions to the current module:

$$I!A \triangleq (x' = x) \wedge (x' \neq x) \\ I!B \triangleq \lambda d : \text{ENABLED } d \\ I!C \triangleq \text{ENABLED } ((\$u' = x) \wedge (\$v' \neq x))$$

Observe that  $I!C$  is not equivalent to  $I!B(I!A)$ . In fact,  $I!C \equiv \text{TRUE}$  and  $I!B(I!A) \equiv \text{FALSE}$ .

We say that instantiation *distributes* over an operator  $Op$  if

$$\overline{Op(e_1, \dots, e_n)} = Op(\overline{e_1}, \dots, \overline{e_n})$$

for any expressions  $e_i$ , where the overlining operator ( $\overline{\phantom{x}}$ ) denotes some arbitrary instantiation. Instantiation distributes over all constant operators—for example,  $+$ ,  $\subseteq$ , and  $\exists$ .<sup>9</sup> Instantiation also distributes over most of the nonconstant operators of TLA<sup>+</sup>, like priming ('') and  $\square$ .

If an operator  $Op$  implicitly binds some identifiers in its arguments, then instantiation would not preserve validity if it distributed over  $Op$ . Our rules for instantiating in an ENABLED expression imply that instantiation does not distribute over ENABLED. It also does not distribute over any operator defined in terms of ENABLED—in particular, the built-in operators WF and SF.

There are two other TLA<sup>+</sup> operators that implicitly bind identifiers: the action composition operator “ $\cdot$ ”, defined in Section 16.2.3, and the temporal operator  $\dot{\Rightarrow}$ , introduced in Section 10.7. The rule for instantiating an expression  $A \cdot B$  is similar to that for ENABLED  $A$ —namely, bound occurrences of variables are replaced by a new symbol. In the expression  $A \cdot B$ , primed occurrences of variables in  $A$  and unprimed occurrences in  $B$  are bound. We handle a formula of the form  $F \dot{\Rightarrow} G$  by replacing it with an equivalent formula in which the quantification is made explicit.<sup>10</sup> Most readers won’t care, but here’s how that equivalent formula is constructed. Let  $\mathbf{x}$  be the tuple  $\langle x_1, \dots, x_n \rangle$  of all declared variables; let  $b, \widehat{x}_1, \dots, \widehat{x}_n$  be symbols distinct from the  $x_i$  and from any bound identifiers in  $F$  or  $G$ ; and let  $\widehat{e}$  be the expression obtained from an expression  $e$  by substituting the variables  $\widehat{x}_i$  for the corresponding variables  $x_i$ . Then  $F \dot{\Rightarrow} G$  is equivalent to

$$(17.8) \quad \begin{aligned} \forall b : ( & (b \in \text{BOOLEAN}) \wedge \square[b' = \text{FALSE}]_b \\ & \wedge \exists \widehat{x}_1, \dots, \widehat{x}_n : \widehat{F} \wedge \square(b \Rightarrow (\mathbf{x} = \widehat{\mathbf{x}})) ) \\ \Rightarrow \exists \widehat{x}_1, \dots, \widehat{x}_n : & \widehat{G} \wedge (\mathbf{x} = \widehat{\mathbf{x}}) \wedge \square[b \Rightarrow (\mathbf{x}' = \widehat{\mathbf{x}}')]_{\langle b, \mathbf{x}, \widehat{\mathbf{x}} \rangle} \end{aligned}$$

Here’s a complete statement of the rules for computing  $\overline{e}$ , for an arbitrary expression  $e$ .

1. Remove all  $\dot{\Rightarrow}$  operators by replacing each subformula of the form  $F \dot{\Rightarrow} G$  with the equivalent formula (17.8).
2. Recursively perform the following replacements, starting from the innermost subexpressions of  $e$ , for each declared variable  $x$  of  $N$ :
  - For each subexpression of the form ENABLED  $A$ , replace each primed occurrence of  $x$  in  $A$  by a new symbol  $\$x$  that is different from any identifier and from any other symbol that occurs in  $A$ .

---

<sup>9</sup>Recall the explanation on pages 320–321 of how we consider  $\exists$  to be a second-order operator. Instantiation distributes over  $\exists$  because TLA<sup>+</sup> does not permit variable capture when substituting in  $\lambda$  expressions.

<sup>10</sup>Replacing ENABLED and “ $\cdot$ ” expressions by equivalent formulas with explicit quantifiers before substituting would result in some surprising instantiations. For example, if  $N$  contains the definition  $E(A) \triangleq \text{ENABLED } A$ , then  $I \triangleq \text{INSTANCE } N$  would effectively obtain the definition  $\text{!}E(A) \triangleq A$ .

- For each subexpression of the form  $B \cdot C$ , replace each primed occurrence of  $x$  in  $B$  and each unprimed occurrence of  $x$  in  $C$  by a new symbol  $\$x$  that is different from any identifier and from any other symbol that occurs in  $B$  or  $C$ .

For example, applying these rules to the inner ENABLED expression and to the “.” expression converts

ENABLED ((ENABLED ( $x' = x$ ))'  $\wedge$  (( $y' = x$ )  $\cdot$  ( $x' = y$ )))

to

ENABLED ((ENABLED ( $\$x' = x$ ))'  $\wedge$  (( $\$y' = x$ )  $\cdot$  ( $x' = \$y$ )))

and applying them again to the outer ENABLED expression yields

ENABLED ((ENABLED ( $\$x' = \$xx$ ))'  $\wedge$  (( $\$y' = x$ )  $\cdot$  ( $\$xx' = \$y$ )))

where  $\$xx$  is some new symbol different from  $x$ ,  $\$x$ , and  $\$y$ .

3. Replace each occurrence of  $q_i$  with  $e_i$ , for all  $i$  in  $1 \dots n$ .

# Chapter 18

## The Standard Modules

Several standard modules are provided for use in TLA<sup>+</sup> specifications. Some of the definitions they contain are subtle—for example, the definitions of the set of real numbers and its operators. Others, such as the definition of  $1 \dots n$ , are obvious. There are two reasons to use standard modules. First, specifications are easier to read when they use basic operators that we’re already familiar with. Second, tools can have built-in knowledge of standard operators. For example, the TLC model checker (Chapter 14) has efficient implementations of some standard modules; and a theorem-prover might implement special decision procedures for some standard operators. The standard modules of TLA<sup>+</sup> are described here, except for the *RealTime* module, which appears in Chapter 9.

### 18.1 Module *Sequences*

The *Sequences* module was introduced in Section 4.1 on page 35. Most of the operators it defines have already been explained. The exceptions are

*SubSeq*( $s, m, n$ ) The subsequence  $\langle s[m], s[m+1], \dots, s[n] \rangle$  consisting of the  $m^{\text{th}}$  through  $n^{\text{th}}$  elements of  $s$ . It is undefined if  $m < 1$  or  $n > \text{Len}(s)$ , except that it equals the empty sequence if  $m > n$ .

*SelectSeq*( $s, \text{Test}$ ) The subsequence of  $s$  consisting of the elements  $s[i]$  such that  $\text{Test}(s[i])$  equals TRUE. For example,

$$\begin{aligned} \text{PosSubSeq}(s) &\triangleq \text{LET } \text{IsPos}(n) \triangleq n > 0 \\ &\quad \text{IN } \text{SelectSeq}(s, \text{IsPos}) \end{aligned}$$

defines  $\text{PosSubSeq}(\langle 0, 3, -2, 5 \rangle)$  to equal  $\langle 3, 5 \rangle$ .

The *Sequences* module uses operators on natural numbers, so we might expect it to extend the *Naturals* module. However, this would mean that any module that extends *Sequences* would then also extend *Naturals*. Just in case someone wants to use sequences without extending the *Naturals* module, the *Sequences* module contains the statement

LOCAL INSTANCE *Naturals*

This statement introduces the definitions from the *Naturals* module, just as an ordinary INSTANCE statement would, but it does not export those definitions to another module that extends or instantiates the *Sequences* module. The LOCAL modifier can also precede an ordinary definition; it has the effect of making that definition usable within the current module, but not in a module that extends or instantiates it. (The LOCAL modifier cannot be used with parameter declarations.)

Everything else that appears in the *Sequences* module should be familiar. The module is in Figure 18.1 on the next page.

## 18.2 Module *FiniteSets*

As described in Section 6.1 on page 66, the *FiniteSets* module defines the two operators *IsFiniteSet* and *Cardinality*. The definition of *Cardinality* is discussed on page 70. The module itself is in Figure 18.2 on the next page.

## 18.3 Module *Bags*

A *bag*, also called a multiset, is a set that can contain multiple copies of the same element. A bag can have infinitely many elements, but only finitely many copies of any single element. Bags are sometimes useful for representing data structures. For example, the state of a network in which messages can be delivered in any order could be represented as a bag of messages in transit. Multiple copies of an element in the bag represent multiple copies of the same message in transit.

The *Bags* module defines a bag to be a function whose range is a subset of the positive integers. An element  $e$  belongs to bag  $B$  iff  $e$  is in the domain of  $B$ , in which case bag  $B$  contains  $B[e]$  copies of  $e$ . The module defines the following operators. In our customary style, we leave unspecified the value obtained by applying an operator on bags to something other than a bag.

$IsABag(B)$       True iff  $B$  is a bag.

$BagToSet(B)$       The set of elements of which bag  $B$  contains at least one copy.

## MODULE Sequences

Defines operators on finite sequences, where a sequence of length  $n$  is represented as a function whose domain is the set  $1 \dots n$  (the set  $\{1, 2, \dots, n\}$ ). This is also how TLA<sup>+</sup> defines an  $n$ -tuple, so tuples are sequences.

LOCAL INSTANCE *Naturals* Imports the definitions from *Naturals*, but doesn't export them.

$\text{Seq}(S) \triangleq \text{UNION } \{[1 \dots n \rightarrow S] : n \in \text{Nat}\}$  The set of all finite sequences of elements in  $S$ .

$\text{Len}(s) \triangleq \text{CHOOSE } n \in \text{Nat} : \text{DOMAIN } s = 1 \dots n$  The length of sequence  $s$ .

$s \circ t \triangleq$  The sequence obtained by concatenating sequences  $s$  and  $t$ .

$[i \in 1 \dots (\text{Len}(s) + \text{Len}(t)) \mapsto \text{IF } i \leq \text{Len}(s) \text{ THEN } s[i] \text{ ELSE } t[i - \text{Len}(s)]]$

$\text{Append}(s, e) \triangleq s \circ \langle e \rangle$  The sequence obtained by appending element  $e$  to the end of sequence  $s$ .

$\text{Head}(s) \triangleq s[1]$  The usual head (first)

$\text{Tail}(s) \triangleq [i \in 1 \dots (\text{Len}(s) - 1) \mapsto s[i + 1]]$  and tail (rest) operators.

$\text{SubSeq}(s, m, n) \triangleq [i \in 1 \dots (1 + n - m) \mapsto s[i + m - 1]]$  The sequence  $\langle s[m], s[m + 1], \dots, s[n] \rangle$ .

$\text{SelectSeq}(s, \text{Test}(\_)) \triangleq$  The subsequence of  $s$  consisting of all elements  $s[i]$  such that  $\text{Test}(s[i])$  is true.

LET  $F[i \in 0 \dots \text{Len}(s)] \triangleq$   $F[i]$  equals  $\text{SelectSeq}(\text{SubSeq}(s, 1, i), \text{Test})$ .

IF  $i = 0$  THEN  $\langle \rangle$

ELSE IF  $\text{Test}(s[i])$  THEN  $\text{Append}(F[i - 1], s[i])$

ELSE  $F[i - 1]$

IN  $F[\text{Len}(s)]$

Figure 18.1: The standard *Sequences* module.

## MODULE FiniteSets

LOCAL INSTANCE *Naturals* Imports the definitions from *Naturals* and *Sequences*, but doesn't export them.

LOCAL INSTANCE *Sequences*

$\text{IsFiniteSet}(S) \triangleq$  A set is finite iff there is a finite sequence containing all its elements.

$\exists \text{seq} \in \text{Seq}(S) : \forall s \in S : \exists n \in 1 \dots \text{Len}(\text{seq}) : \text{seq}[n] = s$

$\text{Cardinality}(S) \triangleq$  Cardinality is defined only for finite sets.

LET  $CS[T \in \text{SUBSET } S] \triangleq$  IF  $T = \{\}$  THEN 0

ELSE  $1 + CS[T \setminus \{\text{CHOOSE } x : x \in T\}]$

IN  $CS[S]$

Figure 18.2: The standard *FiniteSets* module.

|                     |                                                                                                                                                                                                                                                                           |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $SetToBag(S)$       | The bag that contains one copy of every element in the set $S$ .                                                                                                                                                                                                          |
| $BagIn(e, B)$       | True iff bag $B$ contains at least one copy of $e$ . $BagIn$ is the $\in$ operator for bags.                                                                                                                                                                              |
| $EmptyBag$          | The bag containing no elements.                                                                                                                                                                                                                                           |
| $CopiesIn(e, B)$    | The number of copies of $e$ in bag $B$ ; it is equal to 0 iff $BagIn(e, B)$ is false.                                                                                                                                                                                     |
| $B1 \oplus B2$      | The union of bags $B1$ and $B2$ . The operator $\oplus$ satisfies<br>$CopiesIn(e, B1 \oplus B2) = CopiesIn(e, B1) + CopiesIn(e, B2)$ for any $e$ and any bags $B1$ and $B2$ .                                                                                             |
| $B1 \ominus B2$     | The bag $B1$ with the elements of $B2$ removed—that is, with one copy of an element removed from $B1$ for each copy of the same element in $B2$ . If $B2$ has at least as many copies of $e$ as $B1$ , then $B1 \ominus B2$ has no copies of $e$ .                        |
| $BagUnion(S)$       | The bag union of all elements of the set $S$ of bags. For example, $BagUnion(\{B1, B2, B3\})$ equals $B1 \oplus B2 \oplus B3$ . $BagUnion$ is the analog of UNION for bags.                                                                                               |
| $B1 \sqsubseteq B2$ | True iff, for all $e$ , bag $B2$ has at least as many copies of $e$ as bag $B1$ does. Thus, $\sqsubseteq$ is the analog for bags of $\subseteq$ .                                                                                                                         |
| $SubBag(B)$         | The set of all subbags of bag $B$ . $SubBag$ is the analog of SUBSET for bags.                                                                                                                                                                                            |
| $BagOfAll(F, B)$    | The bag analog of the construct $\{F(x) : x \in B\}$ . It is the bag that contains, for each element $e$ of bag $B$ , one copy of $F(e)$ for every copy of $e$ in $B$ . This defines a bag iff, for any value $v$ , the set of $e$ in $B$ such that $F(e) = v$ is finite. |
| $BagCardinality(B)$ | If $B$ is a finite bag (one such that $BagToSet(B)$ is a finite set), then this is its cardinality—the total number of copies of elements in $B$ . Its value is unspecified if $B$ is not a finite bag.                                                                   |

The module appears in Figure 18.3 on the next page. Note the local definition of  $Sum$ , which makes  $Sum$  defined within the *Bags* module but not in any module that extends or instantiates it.

MODULE *Bags*

LOCAL INSTANCE *Naturals* Import definitions from *Naturals*, but don't export them.

$IsABag(B) \triangleq B \in [\text{DOMAIN } B \rightarrow \{n \in \text{Nat} : n > 0\}]$  True iff  $B$  is a bag.

$BagToSet(B) \triangleq \text{DOMAIN } B$  The set of elements at least one copy of which is in  $B$ .

$SetToBag(S) \triangleq [e \in S \mapsto 1]$  The bag that contains one copy of every element of the set  $S$ .

$BagIn(e, B) \triangleq e \in BagToSet(B)$  The  $\in$  operator for bags.

$EmptyBag \triangleq SetToBag(\{\})$

$CopiesIn(e, B) \triangleq \text{IF } BagIn(e, B) \text{ THEN } B[e] \text{ ELSE } 0$  The number of copies of  $e$  in  $B$ .

$B1 \oplus B2 \triangleq$  The union of bags  $B1$  and  $B2$ .

$[e \in (\text{DOMAIN } B1) \cup (\text{DOMAIN } B2) \mapsto CopiesIn(e, B1) + CopiesIn(e, B2)]$

$B1 \ominus B2 \triangleq$  The bag  $B1$  with the elements of  $B2$  removed.

LET  $B \triangleq [e \in \text{DOMAIN } B1 \mapsto CopiesIn(e, B1) - CopiesIn(e, B2)]$   
 IN  $[e \in \{d \in \text{DOMAIN } B : B[d] > 0\} \mapsto B[e]]$

LOCAL  $Sum(f) \triangleq$  The sum of  $f[x]$  for all  $x$  in  $\text{DOMAIN } f$ .

LET  $DSum[S \in \text{SUBSET DOMAIN } f] \triangleq$  LET  $elt \triangleq \text{CHOOSE } e \in S : \text{TRUE}$   
 IN  $\text{IF } S = \{\} \text{ THEN } 0$   
 $\text{ELSE } f[elt] + DSum[S \setminus \{elt\}]$   
 IN  $DSum[\text{DOMAIN } f]$

$BagUnion(S) \triangleq$  The bag union of all elements of the set  $S$  of bags.

$[e \in \text{UNION } \{BagToSet(B) : B \in S\} \mapsto Sum([B \in S \mapsto CopiesIn(e, B)])]$

$B1 \sqsubseteq B2 \triangleq \wedge (\text{DOMAIN } B1) \subseteq (\text{DOMAIN } B2) \wedge \forall e \in \text{DOMAIN } B1 : B1[e] \leq B2[e]$  The subset operator for bags.

$SubBag(B) \triangleq$  The set of all subbags of bag  $B$ .

LET  $AllBagsOfSubset \triangleq$  The set of bags  $SB$  such that  $BagToSet(SB) \subseteq BagToSet(B)$ .  
 $\text{UNION } \{[SB \rightarrow \{n \in \text{Nat} : n > 0\} : SB \in \text{SUBSET } BagToSet(B)]\}$   
 IN  $\{SB \in AllBagsOfSubset : \forall e \in \text{DOMAIN } SB : SB[e] \leq B[e]\}$

$BagOfAll(F(\_), B) \triangleq$  The bag analog of the set  $\{F(x) : x \in B\}$  for a set  $B$ .

$[e \in \{F(d) : d \in BagToSet(B)\} \mapsto Sum([d \in BagToSet(B) \mapsto \text{IF } F(d) = e \text{ THEN } B[d] \text{ ELSE } 0])]$

$BagCardinality(B) \triangleq Sum(B)$  The total number of copies of elements in bag  $B$ .

**Figure 18.3:** The standard *Bags* module.

## 18.4 The Numbers Modules

The usual sets of numbers and operators on them are defined in the three modules *Naturals*, *Integers*, and *Reals*. These modules are tricky because their definitions must be consistent. A module  $M$  might extend both the *Naturals* module and another module that extends the *Reals* module. The module  $M$  thereby obtains two definitions of an operator such as  $+$ , one from *Naturals* and one from *Reals*. These two definitions of  $+$  must be the same. To make them the same, we have them both come from the definition of  $+$  in a module *ProtoReals*, which is locally instantiated by both *Naturals* and *Reals*.

The *Naturals* module defines the following operators:

|                  |                         |     |        |         |                         |
|------------------|-------------------------|-----|--------|---------|-------------------------|
| $+$              | $*$                     | $<$ | $\leq$ | $Nat$   | $\div$ integer division |
| $-$ binary minus | $\wedge$ exponentiation | $>$ | $\geq$ | $\dots$ | $\%$ modulus            |

Except for  $\div$ , these operators are all either standard or explained in Chapter 2. Integer division ( $\div$ ) and modulus ( $\%$ ) are defined so that the following two conditions hold, for any integer  $a$  and positive integer  $b$ :

$$a \% b \in 0 \dots (b - 1) \quad a = b * (a \div b) + (a \% b)$$

The *Integers* module extends the *Naturals* module and also defines the set *Int* of integers and unary minus ( $-$ ). The *Reals* module extends *Integers* and introduces the set *Real* of real numbers and ordinary division ( $/$ ). In mathematics, (unlike programming languages), integers are real numbers. Hence, *Nat* is a subset of *Int*, which is a subset of *Real*.

The *Reals* module also defines the special value *Infinity*. *Infinity*, which represents a mathematical  $\infty$ , satisfies the following two properties:

$$\forall r \in Real : -\text{Infinity} < r < \text{Infinity} \quad -(-\text{Infinity}) = \text{Infinity}$$

The precise details of the number modules are of no practical importance. When writing specifications, you can just assume that the operators they define have their usual meanings. If you want to prove something about a specification, you can reason about numbers however you want. Tools like model checkers and theorem provers that care about these operators will have their own ways of handling them. The modules are given here mainly for completeness. They can also serve as models if you want to define other basic mathematical structures. However, such definitions are rarely necessary for writing specifications.

The set *Nat* of natural numbers, with its zero element and successor function, is defined in the *Peano* module, which appears in Figure 18.4 on the next page. It simply defines the naturals to be a set satisfying Peano's axioms. This definition is separated into its own module for the following reason. As explained in Section 16.1.9 (page 306) and Section 16.1.10 (page 307), the meanings of tuples and strings are defined in terms of the natural numbers. The *Peano* module,

Peano's axioms are discussed in many books on the foundations of mathematics.

— MODULE *Peano* —

This module defines *Nat* to be an arbitrary set satisfying Peano's axioms with zero element *Zero* and successor function *Succ*. It does not use strings or tuples, which in TLA<sup>+</sup> are defined in terms of natural numbers.

$PeanoAxioms(N, Z, Sc) \triangleq$  Asserts that  $N$  satisfies Peano's axioms with zero element  $Z$  and successor function  $Sc$ .  
 $\wedge Z \in N$   
 $\wedge Sc \in [N \rightarrow N]$   
 $\wedge \forall n \in N : (\exists m \in N : n = Sc[m]) \equiv (n \neq Z)$   
 $\wedge \forall S \in \text{SUBSET } N : (Z \in S) \wedge (\forall n \in S : Sc[n] \in S) \Rightarrow (S = N)$

ASSUME  $\exists N, Z, Sc : PeanoAxioms(N, Z, Sc)$  Asserts the existence of a set satisfying Peano's axioms.

$Succ \triangleq \text{CHOOSE } Sc : \exists N, Z : PeanoAxioms(N, Z, Sc)$   
 $Nat \triangleq \text{DOMAIN } Succ$   
 $Zero \triangleq \text{CHOOSE } Z : PeanoAxioms(Nat, Z, Succ)$

**Figure 18.4:** The *Peano* module.

which defines the natural numbers, does not use tuples or strings. Hence, there is no circularity.

As explained in Section 16.1.11 on page 308, numbers like 42 are defined in TLA<sup>+</sup> so that 0 equals *Zero* and 1 equals *Succ[Zero]*, where *Zero* and *Succ* are defined in the *Peano* module. We could therefore replace *Zero* by 0 and *Succ[Zero]* by 1 in the *ProtoReals* module. But doing so would obscure how the definition of the reals depends on the definition of the natural numbers in the *Peano* module.

Most of the definitions in modules *Naturals*, *Integers*, and *Reals* come from module *ProtoReals* in Figure 18.5 on the following two pages. The definition of the real numbers in module *ProtoReals* uses the well-known mathematical result that the reals are uniquely defined, up to isomorphism, as an ordered field in which every subset bounded from above has a least upper bound. The details will be of interest only to mathematically sophisticated readers who are curious about the formalization of ordinary mathematics. I hope that those readers will be as impressed as I am by how easy this formalization is—once you understand the mathematics.

Given the *ProtoReals* module, the rest is simple. The *Naturals*, *Integers*, and *Reals* modules appear in Figures 18.6–18.8 on page 348. Perhaps the most striking thing about them is the ugliness of an operator like  $R!+$ , which is the version of  $+$  obtained by instantiating *ProtoReals* under the name  $R$ . It demonstrates that you should not define infix operators in a module that may be used with a named instantiation.

MODULE *ProtoReals*

This module provides the basic definitions for the *Naturals*, *Integers*, and *Reals* module. It does this by defining the real numbers to be a complete ordered field containing the naturals.

EXTENDS *Peano*

*IsModelOfReals*(*R*, *Plus*, *Times*, *Leq*)  $\triangleq$

Asserts that *R* satisfies the properties of the reals with  $a + b = \text{Plus}[a, b]$ ,  $a * b = \text{Times}[a, b]$ , and  $(a \leq b) = (\langle a, b \rangle \in \text{Leq})$ . (We will have to quantify over the arguments, so they must be values, not operators.)

LET *IsAbelianGroup*(*G*, *Id*,  $\_ + \_$ )  $\triangleq$  Asserts that *G* is an Abelian group with identity *Id* and group operation  $+$ .

$\wedge \text{Id} \in G$   
 $\wedge \forall a, b \in G : a + b \in G$   
 $\wedge \forall a \in G : \text{Id} + a = a$   
 $\wedge \forall a, b, c \in G : (a + b) + c = a + (b + c)$   
 $\wedge \forall a \in G : \exists \text{minusa} \in G : a + \text{minusa} = \text{Id}$   
 $\wedge \forall a, b \in G : a + b = b + a$

$a + b \triangleq \text{Plus}[a, b]$

$a * b \triangleq \text{Times}[a, b]$

$a \leq b \triangleq \langle a, b \rangle \in \text{Leq}$

IN  $\wedge \text{Nat} \subseteq R$

$\wedge \forall n \in \text{Nat} : \text{Succ}[n] = n + \text{Succ}[\text{Zero}]$

$\wedge \text{IsAbelianGroup}(R, \text{Zero}, +)$

$\wedge \text{IsAbelianGroup}(R \setminus \{\text{Zero}\}, \text{Succ}[\text{Zero}], *)$

$\wedge \forall a, b, c \in R : a * (b + c) = (a * b) + (a * c)$

$\wedge \forall a, b \in R : \wedge (a \leq b) \vee (b \leq a)$

$\wedge (a \leq b) \wedge (b \leq a) \equiv (a = b)$

$\wedge \forall a, b, c \in R : \wedge (a \leq b) \wedge (b \leq c) \Rightarrow (a \leq c)$

$\wedge (a \leq b) \Rightarrow \wedge (a + c) \leq (b + c)$

$\wedge (\text{Zero} \leq c) \Rightarrow (a * c) \leq (b * c)$

$\wedge \forall S \in \text{SUBSET } R :$

LET *SBound*(*a*)  $\triangleq \forall s \in S : s \leq a$

IN  $(\exists a \in R : \text{SBound}(a)) \Rightarrow$

$(\exists \text{sup} \in R : \wedge \text{SBound}(\text{sup})$

$\wedge \forall a \in R : \text{SBound}(a) \Rightarrow (\text{sup} \leq a))$

The first two conjuncts assert that *Nat* is embedded in *R*.

The next three conjuncts assert that *R* is a field.

The next two conjuncts assert that *R* is an ordered field.

The last conjunct asserts that every subset *S* of *R* bounded from above has a least upper bound *sup*.

THEOREM  $\exists R, \text{Plus}, \text{Times}, \text{Leq} : \text{IsModelOfReals}(R, \text{Plus}, \text{Times}, \text{Leq})$

*RM*  $\triangleq$  CHOOSE *RM* : *IsModelOfReals*(*RM.R*, *RM.Plus*, *RM.Times*, *RM.Leq*)

*Real*  $\triangleq$  *RM.R*

Figure 18.5a: The *ProtoReals* module (beginning).

We define  $\text{Infinity}$ ,  $\leq$ , and  $-$  so  $-\text{Infinity} \leq r \leq \text{Infinity}$ , for any  $r \in \text{Real}$ , and  $-(-\text{Infinity}) = \text{Infinity}$ .

$$\text{Infinity} \triangleq \text{CHOOSE } x : x \notin \text{Real}$$

$$\text{MinusInfinity} \triangleq \text{CHOOSE } x : x \notin \text{Real} \cup \{\text{Infinity}\}$$

$\text{Infinity}$  and  $\text{MinusInfinity}$  (which will equal  $-\text{Infinity}$ ) are chosen to be arbitrary values not in  $\text{Real}$ .

$$a + b \triangleq \text{RM.Plus}[a, b]$$

$$a * b \triangleq \text{RM.Times}[a, b]$$

$$a \leq b \triangleq \text{CASE } (a \in \text{Real}) \wedge (b \in \text{Real}) \rightarrow \langle a, b \rangle \in \text{RM.Leq}$$

$$\quad \square (a = \text{Infinity}) \wedge (b \in \text{Real} \cup \{\text{MinusInfinity}\}) \rightarrow \text{FALSE}$$

$$\quad \square (a \in \text{Real} \cup \{\text{MinusInfinity}\}) \wedge (b = \text{Infinity}) \rightarrow \text{TRUE}$$

$$\quad \square a = b \rightarrow \text{TRUE}$$

$$a - b \triangleq \text{CASE } (a \in \text{Real}) \wedge (b \in \text{Real}) \rightarrow \text{CHOOSE } c \in \text{Real} : c + b = a$$

$$\quad \square (a \in \text{Real}) \wedge (b = \text{Infinity}) \rightarrow \text{MinusInfinity}$$

$$\quad \square (a \in \text{Real}) \wedge (b = \text{MinusInfinity}) \rightarrow \text{Infinity}$$

$$a/b \triangleq \text{CHOOSE } c \in \text{Real} : a = b * c$$

$$\text{Int} \triangleq \text{Nat} \cup \{\text{Zero} - n : n \in \text{Nat}\}$$

We define  $a^b$  (exponentiation) for  $a > 0$ , or  $b > 0$ , or  $a \neq 0$  and  $b \in \text{Int}$ , by the four axioms

$$a^1 = a \quad a^{m+n} = a^m * a^n \text{ if } a \neq 0 \text{ and } m, n \in \text{Int} \quad 0^b = 0 \text{ if } b > 0 \quad a^{b*c} = (a^b)^c \text{ if } a > 0$$

plus the continuity condition that  $0 < a$  and  $0 < b \leq c$  imply  $a^b \leq a^c$ .

$$a^b \triangleq \text{LET } \text{RPos} \triangleq \{r \in \text{Real} \setminus \{\text{Zero}\} : \text{Zero} \leq r\}$$

$$\text{exp} \triangleq \text{CHOOSE } f \in [(\text{RPos} \times \text{Real}) \cup (\text{Real} \times \text{RPos}) \cup ((\text{Real} \setminus \{\text{Zero}\}) \times \text{Int}) \rightarrow \text{Real}] :$$

$$\quad \wedge \forall r \in \text{Real} : \wedge f[r, \text{Succ}[\text{Zero}]] = r$$

$$\quad \wedge \forall m, n \in \text{Int} : (r \neq \text{Zero}) \Rightarrow$$

$$\quad \quad (f[r, m + n] = f[r, m] * f[r, n])$$

$$\quad \wedge \forall r \in \text{RPos} : \wedge f[\text{Zero}, r] = \text{Zero}$$

$$\quad \wedge \forall s, t \in \text{Real} : f[r, s * t] = f[f[r, s], t]$$

$$\quad \wedge \forall s, t \in \text{RPos} : (s \leq t) \Rightarrow (f[r, s] \leq f[r, t])$$

$$\text{IN } \text{exp}[a, b]$$

**Figure 18.5b:** The *ProtoReals* module (end).

---

MODULE *Naturals*

---

LOCAL  $R \triangleq \text{INSTANCE } \text{ProtoReals}$

$\text{Nat} \triangleq R!\text{Nat}$

$a + b \triangleq a R!+ b$  *R!+ is the operator + defined in module ProtoReals.*

$a - b \triangleq a R!- b$

$a * b \triangleq a R!* b$

$a^b \triangleq a R!^ b$   *$a^b$  is written in ASCII as  $a^b$ .*

$a \leq b \triangleq a R!\leq b$

$a \geq b \triangleq b \leq a$

$a < b \triangleq (a \leq b) \wedge (a \neq b)$

$a > b \triangleq b < a$

$a .. b \triangleq \{i \in R!\text{Int} : (a \leq i) \wedge (i \leq b)\}$

$a \div b \triangleq \text{CHOOSE } n \in R!\text{Int} : \exists r \in 0 .. (b - 1) : a = b * n + r$  *We define  $\div$  and  $\%$  so that  $a = b * (a \div b) + (a \% b)$  for all integers  $a$  and  $b$  with  $b > 0$ .*

$a \% b \triangleq a - b * (a \div b)$

---

**Figure 18.6:** The standard *Naturals* module.

---

MODULE *Integers*

---

EXTENDS *Naturals* *The Naturals module already defines operators like + to work on all real numbers.*

LOCAL  $R \triangleq \text{INSTANCE } \text{ProtoReals}$

$\text{Int} \triangleq R!\text{Int}$

$-. a \triangleq 0 - a$  *Unary - is written -. when being defined or used as an operator argument.*

---

**Figure 18.7:** The standard *Integers* module.

---

MODULE *Reals*

---

EXTENDS *Integers* *The Integers module already defines operators like + to work on all real numbers.*

LOCAL  $R \triangleq \text{INSTANCE } \text{ProtoReals}$

$\text{Real} \triangleq R!\text{Real}$

$a/b \triangleq a R!/ b$   *$R!/$  is the operator / defined in module ProtoReals.*

$\text{Infinity} \triangleq R!\text{Infinity}$

---

**Figure 18.8:** The standard *Reals* module.

# Index

- ' (prime), 16, 82, 312
- ' (prime), double, 322
- '...', 214
- '^, 212
- '^...^', 214, 215, 218–220
- '^, 212, 218
- '^...^', 216, 220
- '., 212
- '. ... .', 215, 218
- " (double quote), 216, 307
- "..." (string), 47, 307, 308
- ^ (exponentiation), 344
- ^+ (BNF operator), 181
- ^\* (BNF operator), 181
- | (BNF operator), 181
- ↪ (function/record constructor), 29, 49, 302, 303
- ██████ (separator), 20
- ██████ (end of module), 21
- \models, 88, 315
- / (division), 73, 344
- \ (set difference), 12, 299, 300
- \\* (end-of-line comment), 32, 288
- \_ (underscore), 46, 285, 318
- (minus), 289, 344
- . , 289
- (overbar), 63, 114, 334, 336, 337
- ¬, *see* negation
- (set of functions), 48, 302, 304
- (step), 16, 312
- (temporal operator), 116, 156, 315, 316, 337
- ~ (stuttering equivalent), 315, 316
- ~', 218
- ~~ (leads to), 91, 314
- ÷ (integer division), 344
- + (plus), 344
- + (BNF operator), 181
- + (suffix operator), 89, 315
- × (Cartesian product), 53, 284, 306, 307
- =, *see* equality
- ≠ (inequality), 300
- ⇒, *see* implies
- ≡, *see* equivalence
- △ (defined to equal), 16, 31
- ≡, 325
- ∞, *see* *Infinity*
- . (syntax element), 284
- . (record field), 28, 180, 305
- .. (integer interval), 20, 344
- .', 218
- , *see* composition of actions
- \* (multiplication), 344
- \* (BNF operator), 181
- (sequence concatenation), 36, 53
- ⊕ (bag union), 342
- ⊖ (bag difference), 342
- (always), 16, 89–90, 288, 314, 315
- (CASE separator), 285
- ◇ (eventually), 91, 288, 314
- < (less than), 344
- ← (substitution), 36
- ≤ (less than or equal), 344
- ⊆ (subset), 12, 299, 300
- ⊑ (bag subset), 342
- (\*...\*) (comment), 32, 288
- $f[e]$  (function application), 301, 303
- $[A]_v$  (action operator), 17, 285, 312
- $[h_1 \mapsto e_1, \dots, h_n \mapsto e_n]$  (record constructor), 29, 305
- $[S \rightarrow T]$  (set of functions), 48, 302, 304

$[x \in S \mapsto e]$  (function constructor), 49, 302, 303  
 $[h_1 : S_1, \dots, h_n : S_n]$  (set of records), 28, 305, 306  
 $\llbracket e \rrbracket$  (meaning), 292, 310  
 $\langle A \rangle_v$  (action operator), 91, 285, 312  
 $\langle e_1, \dots, e_n \rangle$  (tuple), 27, 53, 306  
 $\{ \}$  (empty set), 12, 300  
 $\{ e_1, \dots, e_n \}$  (set), 12, 299, 300  
 $\{x \in S : p\}$  (set constructor), 66, 289, 299, 301  
 $\{e : x \in S\}$  (set constructor), 66, 289, 299, 301  
 $>$  (greater than), 344  
 $\geq$  (greater than or equal), 344  
 $\wedge$ , *see* conjunction  
 $\cap$  (set intersection), 12, 299, 300  
 $\vee$ , *see* disjunction  
 $\cup$  (set union), 12, 299, 300  
 $:$  (in record set constructor), 28  
 $:>$  (function constructor), 249  
 $::=$  (BNF operator), 179  
 $!$  (in EXCEPT), 29, 49, 302, 304–306  
 $!$  (in operator name), 36, 39, 330, 334  
 $?, 325$   
 $\$, 335$   
 $\%$  (modulus), 21, 344  
 $\&$  (BNF operator), 180  
 $\natural, 314, 316$   
 $@$  (in EXCEPT), 29, 302, 304  
 $@@$  (function constructor), 249  
 $\forall$ , *see* universal quantification  
 $\forall$  (temporal universal quantification), 110, 315  
 $\exists$ , *see* existential quantification  
 $\exists_\infty, 94$   
 $\exists_{\text{behavior}}, 316$   
 $\exists_{\text{state}}, 313$   
 $\exists$ , *see* hiding variables  
 $\alpha$  conversion, 319  
 $\beta$  reduction, 319  
 $\varepsilon$ , Hilbert’s, 73, 296  
 $\in$  (syntax element), 284  
    in function expression, 49  
 $\in$  (set membership), 11, 299  
 $\notin$  (not element of), 47, 299, 300  
 $\lambda$  expression, 49, 319

$\mathcal{C}$ -basic, 325  
 $\lambda$  expression, legal, 327  
 $\lambda$  expression, meaning of, 325–327  
 $\lambda$  parameter, 319  
 $0$  (zero), 345  
 $0$ -tuple, 37, 306  
 $1$  (one), 345  
 $1$ -tuple, 306  
  
Abadi, Martín, xvii  
*ABCorrectness* module, 228, 229  
abstraction, 81  
abstraction, choosing an, 24  
abstraction, level of, 76  
action, 16, 312  
    as temporal formula, 89  
composition, *see* composition of actions  
constraint, 241  
execution, 16  
formula, simple, 236  
operator, 269, 312–314  
action parameter, 46  
action, next-state, *see* next-state action  
  
**ACTION-CONSTRAINT** statement, 241  
*ActionConstraint* (in TLC), 241  
actions, commutativity of, 78  
actions, joint, 147, 152  
address, memory, 47  
alignment, 208, 286  
**align0ut** TLATEX option, 217  
alternating bit protocol, 222  
*AlternatingBit* module, 222, 223  
always ( $\square$ ), 16, 89–90, 288, 314, 315  
ambiguity in  $\text{TLA}^+$  grammar, 289  
Analyzer, Syntactic, 207  
and ( $\wedge$ ), *see* conjunction  
and/or, 9  
angle brackets, 27, 96  
*Append*, 36  
application, operator, 320  
argument  
    of a declared constant, 46  
    of a function, 50  
    of an operator, 31  
argument, TLATEX option, 211

- a**ri1 TLC option, 243, 251  
arithmetic operator, 19  
arity of an operator, 318  
arity-correct, 321  
array, 48, 50  
array of clocks, 139  
array, multidimensional, 50  
ASCII representation  
  of characters, 307  
  of reserved words, 19  
  of specification, 19  
  of symbol, 19, 273  
ASCII version of TLA<sup>+</sup>, 275, 305, 307  
ASCII-readable, 220  
*Ass*, 327  
*Assert*, 249  
assertional reasoning, 2  
assignment in TLC, 234  
associativity, 10, 283  
**A**SUME, 42, 327, 332  
  checked by TLC, 241, 261  
assume-guarantee specification, 156  
assumption, 42, 327, 332  
*AsynchInterface* module, 24, 27  
asynchronous interface, 23  
asynchronous system, 3  
atomicity, grain of, 76–78  
aux file, 217  
axioms, Peano’s, 344
- Backus-Naur Form, 179  
bag, 340  
*BagCardinality*, 342  
*BagIn*, 342  
*BagOfAll*, 342  
*Bags* module, 340, 343  
  overridden by TLC, 237  
*BagToSet*, 340  
*BagUnion*, 342  
barrier synchronization, 149  
basic  
  expression, 309  
  state function, 311  
  transition function, 313  
behavior, 15, 18, 314, 315  
  satisfying temporal formula, 18  
behavior, finite, 17, 112, 130  
behavior, Zeno, 120, 128  
behavioral property, 1  
behaviors, equivalence of, 77  
benefits of specification, 75  
binary hour clock, 158  
binary representation, 308  
*BinaryHourClock* module, 159, 160  
blame, 145  
BNF, 179, 276  
*BNFGrammars* module, 179, 183, 184  
BOOLEAN, 46, 293  
Boolean operator, 9, 293, 296  
  precedence, 10  
Boolean operator, simple, 236  
Boolean value, 9  
bound occurrence, 14  
bound variable, 14, 109  
bound, real-time, 122  
bound, strong real-time, 124  
bounded FIFO, 42–43  
bounded quantification, 13, 293  
*BoundedFIFO* module, 43  
box ( $\Box$ ), *see* always  
buffer component, 135, 140  
buffer, FIFO, 35, 140–142  
built-in operator, 20
- C++, 7  
 $C[\![e]\!]$ , 325  
*C*-basic  $\lambda$  expression, 325  
cache, write-through, 54–62, 107–109  
caching memory, 45  
caching memory, real-time, 124–128  
calculator, using TLC as, 261  
capture, variable, 335  
*Cardinality*, 66, 70, 340  
cardinality, 70  
carriage return, 307  
Cartesian product, 53, 306  
CASE, 298  
  evaluated by TLC, 262  
  parsing, 284  
causality, 128  
*CC*, 328  
channel, 28, 97, 99  
*Channel* module, 30  
channel, refining, 159

*ChannelRefinement* module, 161, 162  
 character, 307  
 checkpoint (TLC), 252, 255, 260  
 Chinese, 2  
*Choice*, 295  
 CHOOSE, 47, 73, 294  
     applied to temporal formula, 110  
     evaluated by TLC, 232, 234, 262  
     parsing, 284  
 circular definition, 70  
 class, equivalence, of a state, 246  
 class, Java, 237  
*cleanup* TLC option, 252  
 cleverness, 80  
 clock, hour, *see* hour clock  
 clocks, array of, 139  
 closed-system specification, 44, 167  
 closure, machine, *see* machine closure  
 coarser-grained specification, 76  
 collection of all sets, 66  
 collision, 244, 255  
 comment, 32–34, 82  
     in TLC configuration file, 226  
     shading, 212  
     syntax, 288  
     typeset by TLATEX, 214  
 comment, end-of-line, 32  
 comment, nested, 32  
 comment, space in, 213  
 common subexpression, 60  
 communication, 152  
 commutativity of actions, 78  
 comparable values, 231, 264  
 complete-system specification, 44, 156  
 complexity, relative, 254  
 component, 135  
*CompositeFIFO* module, 142, 143  
 composition of actions ( $\cdot$ ), 77, 312, 313  
     and instantiation, 337  
     evaluated by TLC, 240  
 composition of specifications, 135, 168  
     and liveness, 152  
     and machine closure, 152  
     with disjoint state, 142  
     with shared state, 142–149  
 Composition Rule, 138  
 Compositional Hiding Rule, 155  
 conditional constructs, 298  
**config** TLC option, 253  
*ConfigFileGrammar* module, 262, 263  
 configuration file (TLC), 225, 253, 261  
     grammar, 262  
 conjunct, property, 240  
 conjunct, specification, 240  
 conjunction ( $\wedge$ ), 9, 293  
     applied to temporal formulas, 88  
     as quantification, 13, 105  
     evaluated by TLC, 231  
     of infinitely many formulas, 13  
 Conjunction Rule, SF, 106  
 Conjunction Rule, WF, 105  
 conjuncts, list of, 25, 286, 293  
 connected, strongly, 174  
 connectivity, 173  
 conservative interpretation of Boolean  
     operators, 296  
**CONSTANT**, 25, 46, 327, 329  
 constant  
     declaration, 25  
     level, 324  
     module, 330  
     operator, 268, 291–309  
     parameter, 25, 45  
**CONSTANT** statement, 226  
 constant, internal, 190  
 constant-level expression, 322  
**CONSTANTS**, 25  
**CONSTANTS** statement, 226  
 constraint, 226  
*Constraint* (in TLC), 241  
 constraint, action, 241  
 constraint, global, 140  
 constraint, real-time, 119  
     on disjunction, 128  
 construct, 292  
 constructor, set, 66, 299  
 context, 324–325  
 context, current, 328  
 continually  
     enabled, 106  
     versus continuously, 106  
 continuously  
     enabled, 106, 122  
     versus continually, 106

contract, specification as, 156  
contradiction, proof by, 102  
conversion,  $\alpha$ , 319  
*CopiesIn*, 342  
correct, syntactically, 325  
correctness of module, 332  
Couturier, Dominique, xvii  
**coverage** TLC option, 227, 252, 258  
current context, 328

**d** Syntactic Analyzer option, 207  
data refinement, 164  
data structures, 78–79, 170  
*Dcl*, 327  
deadlock, 222, 251  
**deadlock** TLC option, 242, 251  
debugging, 253  
    variable, 244  
decimal representation, 308  
declaration, 324, 329  
declaration, constant, 25  
declaration, scope of, 31  
declaration, variable, 19  
definition, 31–32, 324  
    of function, 54, 329  
    of operator, 329  
    overriding, 234, 235  
definition, global, 327  
definition, inductive, 69  
definition, local, 170, 327  
definition, module, 324  
delimiter, 285, 286  
**depth** TLC option, 243, 251  
derivative, 176  
diameter of state graph, 254  
difference, set  $(\setminus)$ , 12  
differential equation, 133, 174  
*DifferentialEquations* module, 177, 178  
**difftrace** TLC option, 252, 259  
directed graph, 173  
discrete step, 15  
disjoint-state  
    composition, 142–149  
    specification, 151  
disjunction  $(\vee)$ , 9, 293  
    as quantification, 13, 105  
evaluated by TLC, 231

disjunction  $(\vee)$ , real-time constraint on, 128  
disjuncts, list of, 26, 286, 293  
distribute over, 93, 114, 336  
distributed system, 3  
divide and conquer, 209  
division  
    of integers, 344  
    of real numbers, 73  
**DOMAIN**, 48, 302  
domain of a function, 48, 302  
double quote  $(")$ , 216, 307  
*DR1*, 101  
*DR2*, 102  
dual tautology, 93  
dvi file, 211, 217  
dvips, 213

edge, 172  
element, 11  
empty set, 12  
*EmptyBag*, 342  
**ENABLED**, 97, 312, 313  
    and instantiation, 337  
    evaluated by TLC, 240  
    predicate, computing, 115  
    substitution in, 335  
enabled, 26  
    continually, 106  
    continuously, 106, 122  
    repeatedly, 124  
**ENABLED-free state function**, 311  
end-of-line character, 288, 290  
end-of-line comment, 32  
English, 2, 98  
environment, 43, 147, 156  
equality  $(=)$ , 284, 300  
    of sets, 12  
    versus equivalence, 10, 296  
equation, differential, 133, 174  
equivalence  
    class of a state, 246  
    of specifications, 21  
equivalence  $(\equiv)$ , 9, 293  
    versus equality, 10, 296  
equivalence of behaviors, 77  
error report (TLC), 255

error trace (TLC), 247, 252, 259  
 error, locating, with TLC, 249  
 error, semantic, 208  
 error, syntactic, 208  
 evaluating expressions, 231  
 eventually ( $\diamond$ ), 91, 288, 314  
 eventually always ( $\diamond\Box$ ), 92, 106  
 EXCEPT, 29, 49, 302, 304–306  
 execution of an action, 16  
 execution, terminating, 17  
 existential quantification ( $\exists$ ), 12, 293  
     as disjunction, 13, 105  
     evaluated by TLC, 232  
 existential quantification, temporal, *see*  
     hiding variables  
 explanation of memory scenario, 186  
 expression, 317, 321  
 expression evaluation by TLC, 231  
 expression,  $\lambda$ , *see*  $\lambda$  expression  
 expression, basic, 309  
 expression, level of, 322  
 expression, silly, 67, 222  
     evaluated by TLC, 256  
 EXTENDS, 19, 237, 328  
 EXTENDS, structuring specification with,  
     34

factorial, 54, 67  
 failed to recover state, 247  
 fairness and refinement, 114  
 fairness conditions, combining, 101,  
     105  
 fairness, expressing liveness with, 112  
 fairness, strong, 106–107  
 fairness, weak, 96–100  
     real-time analog, 122  
 FALSE, 9, 293  
*FastSort*, 250  
 field of record, 28  
 FIFO buffer, 35, 140–142  
 FIFO transmission line, 222  
 FIFO, bounded, 42–43  
 file name, root of, 217  
 file, aux, 217  
 file, dvi, 211, 217  
 file, log, 217  
 file, PDF, 211, 212

file, TLC configuration, *see* configura-  
     tion file  
 files, TLAT<sub>E</sub>X output, 217  
 finer-grained specification, 76  
 fingerprint, 244, 247, 255  
 fingers, counting on, 11  
 finite behavior, 112, 130  
*FiniteSets* module, 66, 340, 341  
     overridden by TLC, 237  
 first-order logic, 3, 318  
 flexible variable, 110  
 font of comment, 32  
 font size, 216  
 form feed, 290, 307  
 formal mathematics, 2  
 formal semantics, 292  
 formula, 309, 310  
     splitting, 60  
     used in a sentence, 14  
 formula, simple action, 236  
 formula, temporal, *see* temporal  
     formula  
 formula, valid, 18, 309  
 Frank, Douglas, xvii  
 free occurrence, 14  
 free variable, 14  
 function, 48–51, 72–73, 301  
     expressed with  $:>$  and  $@@$ , 249  
     of multiple arguments, 50, 302  
     transition, 312  
     versus operator, 69–72  
 function definition, 329  
 function, hashing, 244  
 function, nondeterministic, 73  
 function, recursively defined, *see* recur-  
     sive function definition  
 function, state, 25, 310–312  
 functional property, 1

$\mathcal{G}$ , 241, 254  
*GDef*, 327  
 Generalization Rule, 95  
 gibberish, nonsyntactic, 69  
 Gilkerson, Ellen, v  
 global  
     constraint, 140  
     definition, 327

- function definition, 329
- operator definition, 329
- Gonthier, Georges, 307
- grain of atomicity, 76–78
- Grammar*, 180
- grammar, 179
  - of TLA<sup>+</sup>, 276–289
  - of TLC configuration file, 262
- granularity of step, 24, 76–78
- graph, 172
  - graph, directed, 173
  - graph, state ( $\mathcal{G}$ ), *see* state graph
  - graph, undirected, 173
- Graphs* module, 172, 175
- grayLevel** TLATEX option, 213
- Grégoire, Jean-Charles, xvii, 207
- Grover, Vinod, xvii
- Guindon, 1
- handshake protocol, 23
- hashing function, 244
- Hayden, Mark, 221
- Head*, 35, 53
- help** TLATEX option, 212
- henceforth, *see* always
- hexadecimal representation, 308
- hiding variables, 39, 41, 110, 111, 221, 228, 314, 316
  - and composition, 154
- high-level specification, 132, 169
- higher-order logic, 318
- Hilbert’s  $\varepsilon$ , 73, 296
- hoffset** TLATEX option, 217
- hour clock, 15, 96, 98
- hour clock, binary, 158
- hour clock, real-time, 117–121
- HourClock* module, 19, 20
  - with comments, 33
- hybrid specification, 132–133
- identifier, 31, 277, 321
  - of  $\lambda$  parameter, 319
- identifier, run, 252, 255
- IF/THEN/ELSE**, 16
  - evaluated by TLC, 231
  - parsing, 284, 298
- iff, 9
- implementation, 62, 111
  - under interface refinement, 165
- implementation, proving, 62–64
- implementing real-time constraints, 126
- implication, 9
  - as implementation, 62, 111
- implicit substitution (in INSTANCE), 40
- ImpliedAction* (in TLC), 241
- ImpliedInit* (in TLC), 241
- ImpliedTemporal* (in TLC), 241
- implies ( $\Rightarrow$ ), 9, 293
  - definition explained, 10
  - evaluated by TLC, 231
- Implies Generalization Rule, 95
- IN**, *see* LET/IN
- indentation in con/disjunction lists, 26, 286
- inductive definition, 69
- inductive invariant, 61
- infinitely often ( $\square\lozenge$ ), 91, 106
- Infinity*, 122, 344
- infix operator, 270, 278, 345
- info** TLATEX option, 212
- Init* (in TLC), 240
- INIT** statement, 225, 262
- initial predicate, 16
  - evaluated by TLC, 254
- initial state, 16
  - computed by TLC, 237, 240, 241
- InnerFIFO* module, 37, 38
- InnerSequential* module, 200, 201
- InnerSerial* module, 195, 196
- INSTANCE, 36–40, 330–331, 334–338
- INSTANCE, LOCAL, 171
- INSTANCE, structuring specification with, 34
- instantaneous change, 117
- instantiation, 37–40, 330–331, 334–338
- instantiation, parametrized, 39
- Int* (set of integers), 344
- integer division, 344
- integer is real number, 344
- Integers* module, 344, 348
  - overridden by TLC, 237
- Integrate*, 132, 174
- integration, 133
- interface refinement, 158–167

- and implementation, 165  
 and liveness, 165  
 for open-system specifications, 165
- interface, asynchronous, 23  
 interface, memory, 183  
 interface, program, 3, 78  
 interleaving specification, 137, 151  
 internal constant, 190  
 internal variable, 41, 111  
*InternalMemory* module, 52  
 intersection ( $\cap$ ), 12  
 interval, open, 174  
 invariance, proving, 62  
 invariant, 61–62  
     checking by TLC, 225, 242  
     of a specification, 25  
     of an action, 61  
     under stuttering, 90  
*Invariant* (in TLC), 240  
**INVARIANT** statement, 225  
 invariant, inductive, 61  
 invariant, type, 25, 80  
**INVARIANTS** statement, 226  
 irreflexive, 191  
 irreflexive partial order, 71  
*IsABag*, 340  
*IsABehavior*, 316  
*IsAFcn*, 303  
*IsAState*, 313  
*IsDeriv*, 176  
*IsFcnOn*, 140  
*IsFiniteSet*, 43, 66, 340
- Java class, 237  
*JavaTime*, 249  
 Jefferson, David, xvii, 207  
 joint actions, 147, 152  
*JointActionMemory* module, 150
- Kalvala, Sara, xvii  
**key** TLC option, 243  
 keyboard, 81  
 Knuth, Donald Ervin, 19, 211  
 Krishnan, Paddy, xvii
- lambda, *see*  $\lambda$   
 Lamport, Leslie, 211, 221  
 language, 179
- language, programming, 3, 67  
**LATEX**, 211  
**LATEX** commands in **TLATEX**, 219  
**latexCommand** **TLATEX** option, 212  
*LDef*, 327  
 leads to ( $\leadsto$ ), 91, 314  
*LeastGrammar*, 181  
 left associative, 283  
 legal  $\lambda$  expression, 327  
 Leisenring, A. C., 296  
*Len*, 36  
**LET/IN**, 60, 299  
     meaning of, 326  
     parsing, 284  
 level, 321–324  
 level of abstraction, 76  
 level, constant, 324  
 level-correct, 323  
 lexeme, 179  
 lexeme, **TLA+**, 275, 289  
 liberal interpretation of Boolean operators, 296  
 limit, definition of, 177  
 line feed, 307  
 line numbers, 212  
 linearizable memory, 51, 100, 185  
 list of conjuncts, 25, 286, 293  
 list of disjuncts, 26, 286, 293  
 liveness  
     and composition, 152  
     considered unimportant, 116  
     expressed with fairness, 112  
     in interface refinement, 165  
 liveness property, 3, 87  
     checked by TLC, 228, 242, 247  
**LOCAL**, 171, 327, 329, 331, 340  
 local  
     definition, 170, 327  
     function definition, 330  
     **INSTANCE**, 331  
     operator definition, 329  
 log file, 217  
 logic, first-order, 3, 318  
 logic, higher-order, 318  
 logic, predicate, 12–14, 293  
 logic, propositional, 9–11, 293  
 logic, temporal, *see* temporal logic

- logical property, 1  
logical thinking, danger to, 7  
lossy transmission line, 222  
lower-level specification, 169
- machine closure, 111–114, 130, 200, 230  
    and composition, 152
- mapping, refinement, 63, 228
- mathematics, 2, 9, 21, 22, 65, 174
- MaxTime*, 119, 123
- MCAlternatingBit* module, 226, 227
- MDef*, 327
- meaning, 292  
    of  $\lambda$  expression, 325–327  
    of temporal formula, 88
- Melville, Herman, vii
- memory address, 47
- memory interface, 183
- Memory* module, 51, 53
- memory system, 45
- memory value, 47
- memory, caching, 45
- memory, linearizable, 51, 100, 185
- memory, real-time caching, 124–128
- memory, sequentially consistent, 195
- memory, serial, 188
- MemoryInterface* module, 48
- MinTime*, 119, 123
- model, 225
- model checking, 221, 226, 241
- model value, 230, 259
- model, specification as, 76
- moderate interpretation of Boolean operators, 296
- MODULE**, 19
- module, 19  
    correctness, 332  
    definition, 324  
    end, 21  
    finding, 333  
    name, 325  
    overriding, 237  
    unit, 285, 286
- module, constant, 330
- module, meaning of, 317, 327–332
- module, standard, 272, 339
- modulus, 21, 344
- Modus Ponens* Rule, 95
- monolithic specification, 136, 167
- multi-line comment, 214
- multiset, 340
- mutual recursion, 68, 233
- names  
    of modules and operators, 325  
    of sets, 36
- Nat* (set of natural numbers), 12, 21, 308, 344
- natural number, 11
- Naturals* module, 19, 344, 348  
    overridden by TLC, 237
- Nbhd*, 174
- negation ( $\neg$ ), 9, 293  
    of temporal formula, 88
- nested comments, 32
- nested quantifiers, 13
- Next* (in TLC), 240
- NEXT** statement, 225, 264
- next-state action, 16  
    evaluated by TLC, 237
- next-state action, finding error in, 259
- next-state action, invariant of, 61
- next-state action, subaction of, 111
- nice temporal formula, 236
- Nil*, 181
- node, 172
- noEpilog** TLATEX option, 214
- non-machine-closed specification, 200  
    checking, 230
- non-Zeno specification, 130
- nonconstant operator, 309–316
- nondeterministic function, 73
- nondeterministic operator, 73
- noninterleaving specification, 137
- nonsense, 71
- nonsyntactic gibberish, 69
- nonterminal symbol, 179
- noProlog** TLATEX option, 214
- nops** TLATEX option, 213
- not, *see* negation
- now*, 117
- nowarning** TLC option, 253
- NowNext*, 120
- number, 345

**number** TLATEX option, 212  
**number**, natural, 11  
**numbers**, line, 212  
*NZ*, 130  
  
**occurrence**, bound, 14  
**occurrence**, free, 14  
**Ogata**, Kazuhiro, xvii  
**one** (1), 345  
**one-line comment**, 214  
**one-tuple**, 306  
*OneOf*, 182  
**open interval**, 174  
**open set**, 177  
**open-system specification**, 44, 156–158,  
    167  
    interface refinement, 165  
*OpenInterval*, 174  
**operator**, 31, 317  
    application, 320  
    definition, 329  
    name, 325  
    of arithmetic, 19  
    precedence, 271, 283  
    semantics, 291–316  
    symbol, user-definable, 270  
    versus function, 69–72  
**operator**, action, 269, 312–314  
**operator**, arity of, 318  
**operator**, Boolean, 9, 293, 296  
    precedence, 10  
**operator**, built-in, 20  
**operator**, constant, 268, 291–309  
    with arguments, 46  
**operator**, defined in standard module,  
    272  
**operator**, infix, 270, 278, 345  
**operator**, nonconstant, 309–316  
**operator**, nondeterministic, 73  
**operator**, order of, 318  
**operator**, postfix, 270, 278  
**operator**, prefix, 270, 278  
**operator**, recursive definition of, 70  
**operator**, simple Boolean, 236  
**operator**, temporal, 269, 314–316  
**option** (Syntactic Analyzer), 207  
    d, 207  
    s, 207  
**option** (TLATEX), 211  
    alignOut, 217  
    grayLevel, 213  
    help, 212  
    hoffset, 217  
    info, 212  
    latexCommand, 212  
    noEpilog, 214  
    noProlog, 214  
    nops, 213  
    number, 212  
    out, 217  
    ps, 213  
    psCommand, 213  
    ptSize, 216  
    shade, 212  
    style, 218  
    textheight, 216  
    textwidth, 216  
    tlaOut, 218  
    voffset, 217  
**option** (TLC), 251  
    ariI, 243, 251  
    cleanup, 252  
    config, 253  
    coverage, 227, 252, 258  
    deadlock, 242, 251  
    depth, 243, 251  
    difftrace, 252, 259  
    key, 243  
    nowarning, 253  
    recover, 252, 255, 260  
    seed, 251  
    simulate, 251  
    terse, 252  
    workers, 253  
**option argument** (TLATEX), 211  
**or** (∨), *see* disjunction  
**order** of an operator, 318  
**order**, irreflexive partial, 71  
**order**, total, 191  
**OTHER**, 298  
**out** TLATEX option, 217  
**output** of TLC, 253  
**overriding**  
    a definition, 234, 235

- a module, 237
- package, *tlatex*, 218
- Pahalawatta, Kapila, xvii
- Palais, Richard, xvii
- paradox, Russell's, 66
- parameter*,  $\lambda$ , 319
- parameter, action, 46
- parameter, constant, 25, 45
- parametrized instantiation, 39
- parent (in a tree), 174
- parentheses, 10, 284, 287
- parentheses, eliminating, 26
- parsing, 179
- partial order, irreflexive, 71
- path, 173
- PDF file, 211, 212
- Peano* module, 308, 344, 345
- Peano's axioms, 344
- performance, 2
- permutation, 245
- Permutations*, 245, 250
- philosophy, Greek, 128
- Pnueli, Amir, 2
- point (unit of measure), 216
- PosReal*, 174
- possibility, 113
- postfix operator, 270, 278
- PostScript, 211, 212
- power set, 65
- precedence, 10
- of Boolean operator, 10
  - of operators, 271, 283
  - of temporal operators, 92
- range, 283
- predicate logic, 12–14, 293
- predicate, initial, 16
- evaluated by TLC, 254
- predicate, state, *see* state predicate
- prefix operator, 270, 278
- prime ('), 16, 82, 312, 313
- prime ('), double, 322
- primitive operator of semantics, 292
- Print*, 249, 259
- processor, 47
- product, Cartesian, 53, 306
- production (of grammar), 179
- program interface, 3, 78
- programming language, 3, 67
- proof
- by contradiction, 102
  - of implementation, 62–64
  - of invariance, 62
  - rule, temporal, 95
  - rule, TLA, 18
- property checking by TLC, 229, 240
- property conjunct, 240
- PROPERTY statement, 225, 237
- property, behavioral, 1
- property, functional, 1
- property, liveness, 3, 87
- checked by TLC, 228, 242, 247
- property, logical, 1
- property, safety, 3, 87, 153
- propositional logic, 9–11, 293
- protocol, alternating bit, 222
- protocol, handshake, 23
- ProtoReals* module, 308, 344, 346
- ps* TLATEX option, 213
- psCommand* TLATEX option, 213
- pseudo-random number generator, 243, 251
- ptSize* TLATEX option, 216
- quantification, 82
- of temporal formula, 88, 109–110
  - over tuple, 293
- quantification, bounded, 13, 293
- quantification, existential ( $\exists$ ), 12
- quantification, unbounded, 13, 232, 293
- quantification, universal ( $\forall$ ), 12
- quantifier, 12, 293
- nesting, 13
  - parsing, 284
- Quantifier Rule, WF, 105
- queue of unexplored states ( $\mathcal{U}$ ), 241, 254
- quote, double ("), 216, 307
- quotes, single (' ... '), 214
- range of a function, 48
- range, precedence, 283
- Rd*, 186
- reachable state, 226

*Real* (set of real numbers), 67, 308, 344  
 real time, 117–134  
 real-time  
   bound, 122  
   caching memory, 124–128  
   constraint, 119  
     implementing, 126  
     on disjunction, 128  
   constraints, combining, 128  
   hour clock, 117–121  
*Reals* module, 344, 348  
*RealTime* module, 123, 125  
*RealTimeHourClock* module, 119, 121  
 reasoning, assertional, 2  
 receiver, 23, 35, 144, 222  
   component, 135, 140  
 record, 28, 305  
   as function, 49, 50  
   field, 28  
**recover** TLC option, 252, 255, 260  
 recursion, mutual, 68, 233  
 recursive function definition, 54, 67–69,  
   302  
   evaluated by TLC, 233  
 redeclaration, 32  
 reduction,  $\beta$ , 319  
 refinement mapping, 63, 228  
   and fairness, 114  
 refinement of channel, 159  
 refinement, data, 164  
 refinement, interface, *see* interface refinement  
*RegisterInterface* module, 185, 186  
 relation as set of pairs, 191  
 relation, next-state, *see* next-state action  
 relative complexity, 254  
 rely-guarantee specification, 156  
 renaming, instantiation without, 40  
 renaming, parameter, 319  
 repeatedly enabled, 124  
 replacement in TLC, 234, 261  
 request, 47  
 resource, shared, 127  
 response, 47  
 rigid variable, 110  
 root of file name, 217

root of tree, 174  
 round-robin scheduling, 127  
*RTBound*, 122  
*RTMemory* module, 125, 126  
*RTnow*, 120, 123  
*RTWriteThroughCache* module, 129  
 Rudalics, Martin, xvii  
 run identifier, 252, 255  
 running the Syntactic Analyzer, 207  
 running TLATEX, 211  
 running TLC, 251  
 Russell’s paradox, 66  
**s** Syntactic Analyzer option, 207  
 safety property, 3, 87, 153  
 Samborski, Dmitri, 211  
 scenario, 186  
 scheduling shared resource, 127  
 scheduling, round-robin, 127  
 Schreiner, Wolfgang, xvii  
 scope, 31  
 second (unit of time), 117  
 seed, 243, 251, 254  
**seed** TLC option, 251  
*SelectSeq*, 339  
 semantic  
   correctness, 333  
   error, 208  
   part of syntax, 275, 291  
 semantics of TLA<sup>+</sup>, 317  
   deviation by TLC, 262  
 semantics, formal, 292  
 semantics, primitive operator of, 292  
 sender, 23, 35, 144, 222  
   component, 135, 140  
 sentence, 179  
*Seq*, 35  
 sequence, 35, 53, 306  
*Sequences* module, 35, 53, 339, 341  
   overridden by TLC, 237  
 sequentially consistent memory, 195  
 serial memory, 188  
*SerialMemory* module, 195  
 set, 11, 65–66  
   constructor, 66, 299  
   difference (\), 12  
   equality, 12, 300

of all sets, 66  
theory, 3, 11, 43, 300  
set, empty, 12  
set, open, 177  
set, power, 65  
set, symmetry, 246  
set, too big to be, 66  
sets, collection of all, 66  
sets, naming, 36  
*SetToBag*, 342  
SF, 106, 285, 314  
    and instantiation, 337  
SF Conjunction Rule, 106  
**shade**  $\text{TLATE}_{\text{X}}$  option, 212  
shading comments, 212  
shared resource, scheduling, 127  
shared-state  
    composition, 142–149  
    specification, 151  
Shared-State Composition Rule, 146  
silly expression, 67, 222  
    evaluated by TLC, 256  
simple  
    action formula, 236  
    Boolean operator, 236  
    temporal formula, 236  
**simulate** TLC option, 251  
simulation, 226, 243, 251  
simulation, step, 63  
    checked by TLC, 242  
single quotes ('...'), 214  
*Sort*, 235, 250  
*SortSeq*, 250  
sound, 296  
space character, 290, 307  
space in comment, 213  
specification, 1  
    as abstraction, 24  
    as model, 76  
    by scientists, 15  
    conjunction, 240  
    invariant, 25  
    parameter, 25, 45  
    splitting, 34  
    standard form, 221  
    structure, 32–34  
**SPECIFICATION** statement, 225, 237, 264

specification, assume-guarantee, 156  
specification, benefits of, 75  
specification, closed-system, 44, 167  
specification, coarser-grained, 76  
specification, complete-system, 44, 156  
specification, disjoint-state, 151  
specification, *ex post facto*, 228  
specification, finer-grained, 76  
specification, high-level, 132, 169  
specification, hybrid, 132–133  
specification, interleaving, 137, 151  
specification, lower-level, 169  
specification, monolithic, 136, 167  
specification, non-machine-closed, 200  
    checking, 230  
specification, non-Zeno, 130  
specification, noninterleaving, 137  
specification, open-system, 44, 156–158, 167  
specification, rely-guarantee, 156  
specification, shared-state, 151  
specification, Zeno, 128–132  
specifications, composition of, 135, 168  
splitting a specification, 34  
*SRTBound*, 124  
standard form of specification, 221  
standard module, 339  
standard module, operators defined in, 272  
state, 15, 18, 30, 310, 311  
    computed by TLC, 237  
    in  $\mathcal{G}$ , 241  
    printed by TLC, 256  
    specifying universe, 18  
state formula, temporal, 236  
state function, 25, 310–312  
    basic, 311  
state graph ( $\mathcal{G}$ ), 241  
    diameter, 254  
state predicate, 25, 310, 311  
    as temporal formula, 88  
state, initial, 16  
    computed by TLC, 237, 240, 241  
state, reachable, 226  
state, successor, 237  
state-level expression, 322  
step, 16, 151, 312

- satisfying an action, 16
- stuttering, 315
- step simulation, 63
  - checked by TLC, 242
- step, discrete, 15
- step, granularity of, 24, 76–78
- step, stuttering, 17, 90
- STRING, 180, 307, 308
- string, 182, 307
  - represented by TLC, 262
- strong fairness, 106–107
- strong real-time bound, 124
- strongly connected, 174
- structure of a specification, 32–34
- stuttering step, 17, 90, 315
- stuttering, invariant under, 90
- style** TLATeX option, 218
- subaction, 111, 131
- SubBag*, 342
- subgraph, 173
- submodule, 119, 125, 141, 327, 332
- subscript, 96, 153, 285
- SubSeq*, 339
- subsequence operator, 339
- SUBSET, 65, 299, 300
- subset, 12
- substitution, 334
- substitution, implicit (in INSTANCE), 40
- substitution, instantiation as, 37
- succ*, 49
- successor function, 49, 308
- successor state, 237
- symbol, 31
  - user-definable, 270
- symbol, ASCII representation, 19, 273
- symbol, nonterminal, 179
- symbol, terminal, 179
- symmetry, 245, 250
- symmetry set, 246
- SYMMETRY statement, 245, 264
- synchronization, barrier, 149
- Syntactic Analyzer, 207
- syntactic error, 208
- syntactically correct, 325
- syntax, 275
  - syntax, semantic conditions of, 291
- system, 156
- system versus environment, 43
- system, asynchronous, 3
- system, distributed, 3
- system, hybrid, 132
- system, memory, 45
- tab character, 287, 290, 307
- table, truth, 10, 11
- Tail*, 36, 53
- tautology
  - checking with TLC, 261
  - of predicate logic, 13
  - of propositional logic, 11, 296
  - of temporal logic, 92–95
  - versus proof rule, 95
- tautology, dual, 93
- temporal
  - existential quantification, *see* hiding variables
- formula, 18, 88–92, 314
  - checked by TLC, 242
  - evaluated by TLC, 235
  - syntax, 288
  - valid, 315
- formula, meaning of, 88
- formula, nice, 236
- formula, simple, 236
- logic, 2, 116
  - tautology, 92–95
- logic of actions, 2
- operator, 269, 314–316
  - precedence, 92
- proof rule, 95
- state formula, 236
- theorem, 92
  - universal quantification, 110, 315
- Temporal* (in TLC), 240
- temporal-level expression, 322
- terminal symbol, 179
- terminating execution, 17
- termination, 222
- terse** TLC option, 252
- TeX, 19, 211
- textheight** TLATeX option, 216
- textwidth** TLATeX option, 216
- THEOREM, 20, 327, 332
- theorem, 18, 327, 332

- theorem, temporal, 92  
thinking, 83  
*Thm*, 327  
threads used by TLC, 242, 253  
time, real, 117–134  
time, unit of, 117  
*Timer*, 119, 123  
timer, 118  
TLA, 2  
TLA proof rule, 18  
TLA Web page, 1, 19, 63, 207, 211, 221  
TLA<sup>+</sup>, 3  
**tlaout** TLATEX option, 218  
*TLAPlusGrammar* module, 276  
TLATEX, 211, 286, 288  
  output files, 217  
  Web page, 211  
TLATEX, running, 211  
*tlatex* LATEX package, 218  
TLC, 221–264  
*TLC* module, 248  
  overridden by TLC, 237  
TLC value, 230  
TLC, running, 251  
*Tok*, 182  
  *tok*, 180  
token, 180  
tool, TLA<sup>+</sup>, 1, 3, 207  
total order, 191  
trace, TLC error, 247, 252, 259  
transition function, 312  
transition function, basic, 313  
transition-level expression, 322  
transitive, 191  
transmission line, FIFO, 222  
tree, 174  
TRUE, 9, 293  
truth table, 10, 11  
tuple, 27, 53, 306  
tuple, quantification over, 293  
tuple, sequence as, 35  
Tuttle, Mark, xvii, 221  
two-phase handshake, 23  
type in programming language, 67  
type invariant, 25, 80  
type of a variable, 25, 30  
typesetting specifications, 19, 211  
 $\mathcal{U}$ , 241, 254  
uid, 1  
unbounded quantification, 13, 232, 293  
UNCHANGED, 26, 312  
  evaluated by TLC, 238  
undirected graph, 173  
UNION, 65, 299, 300  
union ( $\cup$ ), 12  
unit of time, 117  
unit, module, 285, 286  
universal quantification ( $\forall$ ), 12, 293  
  as conjunction, 13, 105  
  evaluated by TLC, 232  
universal quantification, temporal, 110, 315  
universe specified by state, 18, 43, 135  
unless, 102  
untyped, 30, 67, 296  
URL, 1  
user-definable symbol, 270  
valid formula, 18, 309  
valid temporal formula, 315  
value, 310  
value as set, 43  
value, Boolean, 9  
value, memory, 47  
value, model, 230, 259  
value, TLC, 230  
values, comparable, 231, 264  
VARIABLE, 19, 327, 329  
variable  
  capture, 335  
  declaration, 19  
  hiding, *see* hiding variables  
variable, bound, 14, 109  
variable, debugging, 244  
variable, flexible, 110  
variable, free, 14  
variable, internal, 41, 111  
variable, rigid, 110  
variable, type of, 25, 30  
variable, visible, 41  
VARIABLES, 25  
version of TLC, 221  
view, 243  
VIEW statement, 243, 264

- visible variable, 41
  - voffset** TLATEX option, 217
  - weak fairness, 96–100
    - real-time analog, 122
  - weak fairness, equivalence to strong fairness, 106
  - Web page, TLA, 1, 19, 63, 207, 211, 221
  - Web page, TLATEX, 211
  - WF, 97, 285, 314
    - and instantiation, 337
  - WF Conjunction Rule, 105
  - WF Quantifier Rule, 105
  - WF, equivalence to SF, 106
  - WITH, 36
    - omitted, 40
  - witness, 62
- workers** TLC option, 253
  - Wr*, 186
  - write-through cache, 54–62, 107–109
  - WriteThroughCache* module, 57
  - writing, 1
  - writing, how to start, 24, 79
  - writing, when to start, 83
- Yu, Yuan, xvii
- Zeno behavior, 120, 128
  - Zeno of Elea, 128
  - Zeno specification, 128–132
  - Zermelo-Frankel set theory, 43, 300
  - Zero*, 308
  - zero (0), 345
  - zero-tuple, 37, 306
  - ZF, 43, 300

# Errata to *Specifying Systems*

Leslie Lamport

28 October 2010

These are all the errors and omissions to the first printing (July 2002) of the book *Specifying Systems* reported as of 28 October 2010. Positions in the book are indicated by page and line number, where the top line of a page is number 1 and the bottom line is number  $-1$ . A running head and a page number are not considered to be lines, but all other lines are. Please report any additional errors to the author, whose email address is posted on <http://lamport.org>. The first person to report an error will be acknowledged in any revised edition.

## Uncorrected Errors

### page xiv

The table of contents should list the index, which begins on page 349. [First reported by Dominique Coutourier on 23 August 2002.]

### page 36, line 3 and page 37, line 16

This is not an error, but it would be better if “tail” were replaced by “end”. [First reported by Taj Khattra on 9 December 2004.]

### page 32, line -11

Add the following footnote to that sentence.

However, that section of the specification may not contain  $(*$  or  $*$ ), as in the string constant “ $a*)b$ ”.

[First reported by Damien Doligez on 27 February 2007.]

### page 53

When the error on page 341, line 12 described below is corrected, a side note should be added beside the definition of *Tail* indicating that the actual definition of *Tail* in the *Sequences* module differs from the one given here. [First reported by Dominique Coutourier on 23 August 2002.]

**page 54, line 21**

Readers who want the function *Acker* to equal Ackermann’s function should replace the last THEN expression *Acker*[ $m - 1, 0$ ] with *Acker*[ $m - 1, 1$ ]. [First reported by Jesse Bingham on 16 March 2003.]

**page 56, line 15**

Replace “leave unchanged *vmem*” with “leave unchanged *wmem*”. [First reported by Keith Marzullo on 1 October 2002.]

**page 66, line 20**

Add “page” before “341”.

[First reported by Lásaro Jonas Camargos on 9 August 2004.]

**page 71, lines 18–19**

When the error on page 341, line 12 described below is corrected, the phrase “The definition of *Tail*” should be changed to “The definition of *Tail* given above”. [First reported by Dominique Coutourier on 23 August 2002.]

**page 89, line 4**

Replace “pair of steps” by “pair of states”. [First reported by Matthieu Lemerrer on 28 October 2010.]

**page 96, line –5**

Replace  $\langle Hnxt \rangle_{hr}$  by  $\langle HCnxt \rangle_{hr}$ .

[First reported by Santiago Zanella Béguin on 11 June 2003.]

**page 100, line 17**

Replace “instead of (8.7)” by “instead of (8.6)”. [First reported by Lucio Nardelli on 11 July 2010.]

**page 115, lines –1 and –2**

Replace “rdy” by “busy”.

[First reported by Casey Klein on 15 February 2010.]

**page 140, line -8**

Replace  $N(k)$  with  $N_k$ . [First reported by Rodrigo Schmidt on 9 November 2006.]

**page 189, line -7**

Add the missing space between “*opId*” and “to”.

[First reported by Rodrigo Schmidt on 9 November 2006.]

**page 234, line 8**

Replace “on page 14.5.3” with “on page 261”.

[First reported by William A. Welch on 25 July 2002.]

**page 237, Section 14.2.5**

The description of overriding should state that a Java class need not override all the definitions in a module. The definitions of operators not overridden are taken from the module. [First reported by Yuan Yu on 31 May 2002.]

**page 243, line -13**

Replace “*key*” with “*seed*”. [First reported by Simon Zambrovski on 21 April 2009.]

**page 252, line -13**

Delete “*num*”. [First reported by Jesse Bingham on 15 June 2004.]

**page 256, footnote**

Delete “though unlikely,”. [First reported by Jesse Bingham on 15 June 2004.]

**page 278, definition of *InfixOp***

The set of *InfixOp* token strings should include “`<=`” and “`\notin`”. [First reported by Damien Doligez on 28 February 2007.]

**page 280ff**

The BNF production for *G.Expression* is missing this alternative:

| *G.Expression* & *tok*(“.”) & *Name*

**page 289, line 15**

This isn’t an error, but the various possibilities would be better illustrated if this line were replaced by

INSTANCE *M* WITH `+`  $\leftarrow$  *Plus*, `Minus`  $\leftarrow$  `-`

[First reported by Damien Doligez on 19 March 2007.]

## page 293

The formula

$$\exists x, y \in S, z \in T : p \triangleq \exists x \in S : (\exists y \in S : (\exists z \in T : p))$$

is incorrect. It should state that the formulas on either side of the  $\triangleq$  are equivalent if  $S$  and  $T$  contain no occurrences of  $x$ ,  $y$ , or  $z$ . [First reported by Stephan Merz on 7 November 2006.]

## page 303, bottom of page

To my surprise (and I think to his too), Stephan Merz discovered that the rules for the operator  $[x \in S \mapsto e]$  are incomplete and the following additional rule should be added.

$$IsAFcn([x \in S \mapsto e])$$

[First reported by Stephan Merz on 2 August 2005.]

## page 304, line 4

This “ $\triangleq$ ” relation holds only for  $n > 1$ .

## page 307, line -11

The list of characters that can appear in a string should include “!”.

[First reported by Damien Doligez on 26 March 2007.]

## page 316, line 11

The definition of  $\sim_x$  should be

$$\sigma \sim_x \tau \triangleq \natural\sigma = [n \in \text{DOMAIN } \natural\tau \mapsto (\natural\tau[n])_{x \leftarrow \natural\sigma[n][\![x]\!]}]$$

[First reported by Raymond Boute on 5 October 2005.]

## Section 17.4 (page 325ff)

The rules for defining the meaning of a  $\lambda$  expression do not prevent “variable capture” in all cases. A more sophisticated definition is needed. [First reported by Georges Gonthier on 9 May 2007.]

## page 326, line -10

Between “ $e$  is” and “where”, add “LET  $Op \triangleq d$  IN  $exp$ ”.

[First reported by Rodrigo Schmidt on 9 November 2006.]

## page 332, Section 17.5.6

The two forms

$$\text{THEOREM } Op \triangleq \exp \quad \text{and} \quad \text{ASSUME } Op \triangleq \exp$$

are not part of the language; all reference to them should be omitted.

[First reported by Rodrigo Schmidt on 9 November 2006.]

## pages 337–338

The two bulleted subitems of item 2 at the bottom of page 337 should be modified so that, before doing the indicated substitutions in  $A$ ,  $B$ , and  $C$ , the following substitutions are performed to their subexpressions, for any  $e$  and  $v$ :

$$\begin{array}{ll} \text{UNCHANGED } v & \rightarrow v' = v \\ [e]_v & \rightarrow e \vee (v' = v) \\ \langle e \rangle_v & \rightarrow e \wedge (v' \neq v) \end{array}$$

[First reported by Yuan Yu on 1 October 2002.]

## page 341, line 12

The definition of *Tail* in the *Sequences* module defines the tail of the empty sequence to be the empty sequence. The tail of the empty sequence should be left unspecified. One possible definition is:

$$\begin{aligned} \text{Tail}(s) \triangleq & \text{ IF } \text{Len}(s) \neq 0 \text{ THEN } [i \in 1 \dots (\text{Len}(s) - 1) \mapsto s[i + 1]] \\ & \text{ELSE CHOOSE } n : \text{FALSE} \end{aligned}$$

## page 345, module *Peano*

The definition of *PeanoAxioms* needs the following additional conjunct asserting that the function *Sc* is injective.

$$\wedge \forall m, n \in N : (\text{Sc}[m] = \text{Sc}[n]) \Rightarrow (m = n)$$

[First reported by Stephan Merz on 3 August 2005.]

## page 347, line 17

In this comment, “continuity condition” should be changed to “monotonicity condition”. [First reported by Peter Hancock on 13 August 2002.]

## Index

The index should contain the entries

octal representation, 308  
numbers, representation of, 308

The entry for “string” should also include a reference to page 47.

# How to Write a 21<sup>st</sup> Century Proof

Leslie Lamport

23 November 2011  
Minor change on 15 January 2012

## **Abstract**

A method of writing proofs is described that makes it harder to prove things that are not true. The method, based on hierarchical structuring, is simple and practical. The author's twenty years of experience writing such proofs is discussed.

## Contents

|          |                                               |           |
|----------|-----------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                           | <b>1</b>  |
| <b>2</b> | <b>An Example</b>                             | <b>2</b>  |
| <b>3</b> | <b>Hierarchical Structure</b>                 | <b>6</b>  |
| <b>4</b> | <b>A Language for Structured Proofs</b>       | <b>9</b>  |
| 4.1      | The Proof Steps of TLA <sup>+</sup> . . . . . | 9         |
| 4.2      | Hierarchical Numbering . . . . .              | 13        |
| 4.3      | Equational Proofs . . . . .                   | 14        |
| 4.4      | Comments . . . . .                            | 14        |
| 4.5      | Completely Formal Proofs . . . . .            | 15        |
| <b>5</b> | <b>Experience</b>                             | <b>16</b> |
| <b>6</b> | <b>Objections to Structured Proof</b>         | <b>18</b> |
| <b>7</b> | <b>Beginning</b>                              | <b>19</b> |
|          | <b>Acknowledgement</b>                        | <b>19</b> |
|          | <b>References</b>                             | <b>19</b> |
|          | <b>Appendix: A Formal Proof</b>               | <b>21</b> |

In addition to developing the students' intuition about the beautiful concepts of analysis, it is surely equally important to persuade them that precision and rigor are neither deterrents to intuition, nor ends in themselves, but the natural medium in which to formulate and think about mathematical questions.

Michael Spivak, *Calculus* [7]

## 1 Introduction

Some 20 years ago, I published an article titled *How to Write a Proof* in a festschrift in honor of the 60<sup>th</sup> birthday of Richard Palais [5]. In celebration of his 80<sup>th</sup> birthday, I am describing here what I have learned since then about writing proofs and explaining how to write them.

As I observed in the earlier article, mathematical notation has changed considerably in the last few centuries. Mathematicians no longer write formulas as prose, but use symbolic notation such as  $e^{i\pi} + 1 = 0$ . On the other hand, proofs are still written in prose pretty much the way they were in the 17<sup>th</sup> century. The proofs in Newton's *Principia* seem quite modern. This has two consequences: proofs are unnecessarily hard to understand, and they encourage sloppiness that leads to errors.

Making proofs easier to understand is easy. It requires only the simple application of two principles: structure and naming. When one reads a sentence in a prose proof, it is often unclear whether the sentence is asserting a new fact or justifying a previous assertion; and if it asserts a new fact, one has to read further to see if that fact is supposed to be obviously true or is about to be proved. Adding structure makes it clear what the function of each assertion is. If an assertion follows from previously stated facts, in a prose proof it is often unclear what those facts are. Naming all facts makes it easy to tell the reader exactly which ones are being used.

Introducing structure has another important benefit. When writing a proof, we are continually deciding how detailed an explanation to provide the reader. Additional explanation helps the reader understand what is being shown at that point in the proof. However, in a prose proof, the additional length makes it harder to follow the logic of the complete proof. Proper structuring allows us to add as much detailed explanation as we like without obscuring the larger picture.

Eliminating errors from proofs is not as easy. It takes precision and rigor, which require work. Structuring proofs makes it possible to avoid errors; hard work is needed to make it probable.

One mistake I made in the earlier article was to advocate making proofs both easier to read and more rigorous. Learning both a new way to write proofs and how to be more precise and rigorous was too high a barrier for most mathematicians. I try here to separate the two goals. I hope that structuring proofs to be easier to read will make mathematicians more aware of how sloppy their proofs are and encourage greater precision and rigor. But the important goal is to stop writing 17<sup>th</sup> century prose proofs in the 21<sup>st</sup> century.

Another mistake I made was giving the impression that I know the best way of writing proofs. I claim to have a *better* way to write proofs than mathematicians now use. In the 21<sup>st</sup> century, it is not hard to improve on 17<sup>th</sup> century proofs. What I describe here is what I have learned in over 20 years of writing structured proofs. I am sure my way of writing proofs can be improved, and I encourage mathematicians to improve it. They will not do it by remaining stuck in the 17<sup>th</sup> century.

The way I now write proofs has profited by my recent experience designing and using a formal language for machine-checked proofs. Mathematicians will not write completely formal proofs in the next 20 years. However, learning how to write a formal proof can teach how to write ordinary, informal ones. Formal proofs are therefore briefly discussed in Section 4, and the appendix contains a formal, machine-checked proof of the example introduced in Section 2.

I am writing this at the end of an era. For millenia, mathematics has been recorded on the processed remains of dead plants and animals. By the end of the 21<sup>st</sup> century, mathematical books and articles will be read almost exclusively on electronic devices—or perhaps using even more advanced technology. This will enable the use of hypertext, which is the natural medium for structured proofs. However, in the next decade or two, mathematics will still be printed on paper or disseminated as static electronic images of printed pages. I therefore concentrate on how to write proofs as conventional printed text, with only brief mention of hypertext.

## 2 An Example

To illustrate how to structure a proof, let us convert a simple prose proof into a structured one that is easier to read. The example I have chosen is adapted from the proof of a corollary to the Mean Value Theorem from Spivak's *Calculus* [7, page 170]; it is shown in Figure 1. (The original corollary covered both the case of increasing and decreasing functions, but Spivak proved only

**Corollary** If  $f'(x) > 0$  for all  $x$  in an interval, then  $f$  is increasing on the interval.

**Proof** Let  $a$  and  $b$  be two points in the interval with  $a < b$ . Then there is some  $x$  in  $(a, b)$  with

$$f'(x) = \frac{f(b) - f(a)}{b - a}.$$

But  $f'(x) > 0$  for all  $x$  in  $(a, b)$ , so

$$\frac{f(b) - f(a)}{b - a} > 0.$$

Since  $b - a > 0$  it follows that  $f(b) > f(a)$ .  $\blacksquare$

Figure 1: Spivak's corollary and proof.

the increasing case; that proof is copied verbatim.) For compactness, I refer to the corollary and proof as Spivak's.

I chose this proof because it is short and simple. I would not normally structure such a simple four-sentence proof because the mathematically sophisticated readers of my papers would have no trouble understanding it. However, Spivak was writing for first-year calculus students. Moreover, seeing that even this simple proof is not so easy to understand should make it clear that structuring is needed for more complicated proofs.

We structure this proof as a sequence of named statements, each with a proof, and we adopt the conventional approach of using sequential numbers as statement names. The first numbered statement is the first sentence of the prose proof:

1. Let  $a$  and  $b$  be two points in the interval with  $a < b$ .

When we choose something in a proof, we have to prove that it exists. The proof of this statement should therefore prove the existence of points  $a$  and  $b$  in the interval with  $a < b$ . However, this can't be proved. The corollary assumes an interval, but it doesn't assume that the interval contains two different points. The book's definition of an interval allows a single point and the empty set as intervals. We can't prove that the interval contains two distinct points because it needn't.

A mathematician, examining the entire proof, will realize that we don't really have to prove that the interval contains two distinct points. However, this is not obvious to a beginning calculus student. A proof is not easy

to understand if we must read the entire proof to understand why its first sentence is justified. Let us just note this problem for now; we'll fix it later.

The second statement of the structured proof comes from the second sentence of the prose proof:

$$2. \text{ There is some } x \text{ in } (a, b) \text{ with } f'(x) = \frac{f(b) - f(a)}{b - a}.$$

Spivak gives no justification for this assertion; the “Then” that begins the sentence alerts readers that it follows from facts that preceded it. The reader must deduce that those facts are:  $a$  and  $b$  are in the interval,  $f$  is differentiable on the interval, and the Mean Value Theorem. Instead of forcing the reader to deduce them, we make the proof easier to understand by stating these facts in the following proof of the statement:

PROOF: By 1, the corollary's hypothesis, and the Mean Value Theorem.

The prose proof's third sentence, “But  $f'(x) > 0 \dots$ ” asserts two facts, so let's turn them into steps 3 and 4 of the structured proof. The final sentence could also be viewed as two statements, but I find it more natural to turn it into the assertion  $f(b) > f(a)$  and its proof. Adding these statements and their proofs, we obtain the simple structured proof of Figure 2.

Examine the structured version. Except for the  $b - a > 0$  in the proof of step 5, all the explanation in the steps' proofs is missing from the original. Adding that explanation makes the proof easier to understand. It might seem excessive to a mathematician, who can easily fill in those missing justifications. However, a web search reveals that Spivak's text, while highly regarded for its rigor, is considered very difficult for most students. Those students would appreciate the additional explanation.

Adding the explanations missing from Spivak's proof does make the structured proof somewhat longer, occupying about 40% more vertical space in the typeset version I am now viewing. However, the extra length does not obscure the proof's structure. It is easy to ignore the subproofs and read just the five steps. I find the structure more apparent in the rewritten version than in the original, and I think others will too—especially once they get used to reading structured proofs.

The advantage of the structured version becomes much more obvious with hypertext. When books are read as hypertext, readers will first see the corollary with no proof. They will click or tap on the corollary or some adjacent icon to view the proof. With the structured version, they will then see just the five steps—essentially the same proof as Spivak's, except with

**Corollary** If  $f'(x) > 0$  for all  $x$  in an interval, then  $f$  is increasing on the interval.

1. Let  $a$  and  $b$  be two points in the interval with  $a < b$ .

PROOF: ???

2. There is some  $x$  in  $(a, b)$  with  $f'(x) = \frac{f(b) - f(a)}{b - a}$ .

PROOF: By 1, the corollary's hypothesis, and the Mean Value Theorem.

3.  $f'(x) > 0$  for all  $x$  in  $(a, b)$ .

PROOF: By the hypothesis of the corollary and 1.

4.  $\frac{f(b) - f(a)}{b - a} > 0$

PROOF: By 2 and 3.

5.  $f(b) > f(a)$

PROOF: By 1, which implies  $b - a > 0$ , and 4.

Figure 2: Adding structure to Spivak's proof.

the omission of “Since  $b - a > 0$  it follows that”. They can then view and hide the proofs of individual steps as they choose. (They can also choose to view the complete proof.)

We are not through with the proof. We cannot prove step 1. This problem is a symptom of a glaring omission in the proof. Has the reader observed that neither Spivak's proof nor its structured version provides the slightest hint of why the proof actually proves the corollary? Readers must figure that out by themselves. Leaving such an important step to the reader does not make the proof easy to understand.

This gaffe is impossible in the proof style that I propose. The style requires that the last step of every proof be a statement of what it is that the proof is trying to prove—in this case, the statement of the corollary. Instead of repeating what we're trying to prove, which has already been stated, we write Q.E.D. (that which was to be shown) instead. A structured proof of the corollary cannot end as in Figure 2, with an assertion that is not what we are trying to prove.

The reader may have surmised that what is missing from Spivak's proof is an explanation of why we should choose  $a$  and  $b$ . His first sentence, and our first step, should have been:

If suffices to assume that  $a$  and  $b$  are two points in the interval with  $a < b$  and prove  $f(b) > f(a)$ .

The justification of that step is that it follows from the definition of an increasing function. Step 5 can then be replaced by:

### 5. Q.E.D.

PROOF: Step 1 implies  $b - a > 0$ , so 4 implies  $f(b) - f(a) > 0$ , which implies  $f(b) > f(a)$ . By 1, this proves the corollary.

We can further improve the proof by making more use of naming. Let's give the nameless interval in the statement of the corollary the name  $I$ . Also, step 1 asserts two assumptions: that  $a$  and  $b$  are in  $I$  and that  $a < b$ . Let's name those assumptions so we can refer in the proof to the exact one that is being used. The result is the proof in Figure 3.

Transforming Spivak's proof to the structured proof in Figure 3 was quite simple. We could have done it differently—for example, by making  $b - a > 0$  a separate step, or by removing a bit of the explanation and using this proof of step 5:

PROOF: Assumption 1.2 implies  $b - a > 0$ , so 4 implies  $f(b) > f(a)$ .  
By 1, this proves the corollary.

With any sensible choices, the resulting structured proof would be easier to understand than Spivak's unstructured proof.

## 3 Hierarchical Structure

Writing the structured proof of Figure 3 forced us to write a justification for each step. Having to do this helps catch errors. However, it is not enough to ensure error-free proofs. In fact, our structured proof contains an important omission.

The best way I know to eliminate errors is to imagine that there is a curious child sitting next to us. Every time we write an assertion, the child asks: *Why?* When we wrote the proof of step 2, the child would ask “Why does step 2 follow from the assumptions and the Mean Value Theorem?” To answer, we would point to Spivak's statement of the Mean Value Theorem:<sup>1</sup>

---

<sup>1</sup>The astute reader will notice that this theorem assumes the unstated hypothesis  $a < b$ . When introducing the notation  $(a, b)$ , the book states that  $a < b$  “is almost always assumed (explicitly if one has been careful, implicitly otherwise).”

**Corollary** If  $f'(x) > 0$  for all  $x$  in an interval  $I$ , then  $f$  is increasing on  $I$ .

1. It suffices to assume

1.  $a$  and  $b$  are points in  $I$
2.  $a < b$

and prove  $f(b) > f(a)$ .

PROOF: By definition of an increasing function.

2. There is some  $x$  in  $(a, b)$  with  $f'(x) = \frac{f(b) - f(a)}{b - a}$ .

PROOF: By assumptions 1.1 and 1.2, the hypothesis that  $f$  is differentiable on  $I$ , and the Mean Value Theorem.

3.  $f'(x) > 0$  for all  $x$  in  $(a, b)$ .

PROOF: By the hypothesis of the corollary and assumption 1.1.

4.  $\frac{f(b) - f(a)}{b - a} > 0$

PROOF: By 2 and 3.

5. Q.E.D.

PROOF: Assumption 1.2 implies  $b - a > 0$ , so 4 implies  $f(b) - f(a) > 0$ , which implies  $f(b) > f(a)$ . By 1, this proves the corollary.

Figure 3: Fixing the proof.

**Theorem 4 (The Mean Value Theorem)** If  $f$  is continuous on  $[a, b]$  and differentiable on  $(a, b)$ , then there is a number  $x$  in  $(a, b)$  such that

$$f'(x) = \frac{f(b) - f(a)}{b - a}$$

Step 2 is identical to the conclusion of the theorem, but the child would ask why the hypotheses hold. In particular, why is  $f$  continuous on  $[a, b]$ ? We would answer, “Because  $f$  is differentiable on  $I$ .” “But why does that imply  $f$  is continuous?” We would turn back 36 pages in the book and point to Spivak’s Theorem 1, which asserts that differentiability implies continuity. “You didn’t tell me you needed Theorem 1.” The child is right. If that result is stated as a numbered theorem, surely its use should be mentioned. The proof of step 2 should be:

PROOF: By the Mean Value Theorem and 1.2, since 1.1 and the hypothesis of the corollary imply that  $f$  is differentiable on  $[a, b]$ , so Theorem 1 implies it is continuous on  $[a, b]$ .

How much detail is necessary? For example, why do 1.1 and the hypothesis of the corollary, which asserts that  $f$  is differentiable on  $I$ , imply that  $f$  is differentiable on  $[a, b]$ ? The proof is assuming the fact that  $a$  and  $b$  in the interval  $I$  implies that  $[a, b]$  is a subset of  $I$ . Should this also be mentioned?

If you are writing the proof to show someone else that the theorem is correct, then the answer depends on the sophistication of the reader. A beginning student needs more help understanding a proof than does a mathematician.

If you are writing the proof for yourself to make sure that the theorem is correct, then the answer is simple: if the truth of a statement is not completely obvious, or if you suspect that there may be just the slightest possibility that it is not correct, then more detail is needed. When you write a proof, you believe the theorem to be true. The only way to avoid errors is to be ruthlessly suspicious of everything you believe. Otherwise, your natural desire to confirm what you already believe to be true will cause you to miss gaps in the proof; and every gap could hide an error that makes the entire result wrong.

Our proof of step 2 is a prose paragraph. As with any prose proof, every detail we add to it makes it harder to follow. In ordinary mathematical writing, the only solution to this problem would be to state and prove the step as a separate lemma. However, making each such subproof a lemma would submerge the interesting results in a sea of lemmas. With structured proofs there is a simple solution: replace the paragraph with a structured proof. Figure 4 shows the result of doing this for the proof of step 2 above.

2. There is some  $x$  in  $(a, b)$  with  $f'(x) = \frac{f(b) - f(a)}{b - a}$ .

- 2.1.  $f$  is differentiable on  $[a, b]$ .

PROOF: By 1.1, since  $f$  is differentiable on  $I$  by hypothesis.

- 2.2.  $f$  is continuous on  $[a, b]$ .

PROOF: By 2.1 and Theorem 1.

- 2.3. Q.E.D.

PROOF: By 2.1, 2.2, and the Mean Value Theorem.

Figure 4: An expanded proof of step 2.

With hypertext, there is no problem adding extra detail like this. We could add enough levels to reduce the reasoning to applications of elementary axioms. The reader can stop opening lower levels of the proof when satisfied that she understands why the statement is true. With proofs on paper, extra details kill trees. But while not as convenient as hypertext, the use of indentation makes it easy for the reader to skip over details that do not interest her.

## 4 A Language for Structured Proofs

TLA<sup>+</sup> is a formal language designed for specifying and reasoning about algorithms and computer systems [3]. It includes a standard formalization of ordinary mathematics based on first-order logic and Zermelo-Fraenkel set theory. TLA<sup>+</sup> contains constructs for writing proofs that formalize the style of structured proofs that I advocate. The TLA<sup>+</sup> Toolbox is a program with a graphical interface for writing and checking TLA<sup>+</sup> specifications and proofs. It provides the type of hypertext viewer of structured proofs that I expect eventually to be commonplace.

This section describes the TLA<sup>+</sup> proof constructs that are relevant for ordinary mathematics. With one possible exception, these constructs can be written informally so that their meanings are obvious to a reader who has never before seen structured proofs. However, it is as silly to express logical proof structures in words as it is to express equations in words. When mathematicians leave the 17<sup>th</sup> century and begin writing structured proofs, I trust that they will adopt compact notation to replace phrases like “We now consider the case in which.”

### 4.1 The Proof Steps of TLA<sup>+</sup>

#### Simple Assertions

The most common type of proof step is a simple assertion. Steps 2–5 of the proof in Figure 3 are such assertions. A simple assertion is a mathematical formula. For example, the statement of the corollary can be written in TLA<sup>+</sup> as:

$$(1) \quad \forall f : \forall I \in SetOfIntervals : \\ (\forall x \in I : d(f)[x] > 0) \Rightarrow IsIncreasingOn(f, I)$$

where *SetOfIntervals* is defined to be the set of all intervals of real numbers,  $d(f)$  is the derivative<sup>2</sup> of  $f$ , square brackets are used for function application,  $\Rightarrow$  is logical implication, and *IsIncreasingOn* is defined by

$$\text{IsIncreasingOn}(f, I) \triangleq \forall x, y \in I : (x < y) \Rightarrow (f[x] < f[y])$$

If asked to formalize the statement of the corollary, most mathematicians would probably write something like (1), except with unimportant notational differences.

A Q.E.D. step is a simple assertion of the formula that is the proof's current goal.

#### ASSUME/PROVE

The statement of the corollary is actually not a simple assertion. We can replace  $f$  and  $I$  by other variables without changing the meaning of the formula (1). However, if we changed  $f$  and  $I$  just in the Corollary's statement, then Spivak's proof would make no sense because the variables  $f$  and  $I$  that appear in it would be meaningless. The statement of the corollary can be expressed in TLA<sup>+</sup> as this ASSUME/PROVE statement:

```
ASSUME NEW f, NEW I ∈ SetOfIntervals, ∀x ∈ I : d(f)[x] > 0
PROVE IsIncreasingOn(f, I)
```

This ASSUME/PROVE asserts the truth of formula (1). It also declares the goal of the corollary's proof to be the PROVE formula and allows the assumptions in the ASSUME clause to be assumed in the proof. The assumption NEW  $x$  introduces a new variable  $x$ , implicitly asserting that the conclusion is true for all values of  $x$ . The assumption NEW  $x \in S$  is equivalent to the two assumptions NEW  $x$ ,  $x \in S$ .

An ASSUME/PROVE can appear as a proof step as well as the statement of the theorem. The PROVE formula is the goal of the step's proof, and the ASSUME clause's assumptions can be used only in that proof. We can consider an ordinary assertion to be an ASSUME/PROVE with an empty ASSUME clause.

It is not obvious how to write an ASSUME/PROVE step in ordinary mathematical prose. When an assumption is introduced in a prose proof, it is assumed to hold until some unspecified later point in the proof. (One reason prose proofs are hard to understand is that it can be difficult to figure out the scope of an assumption.) As in the statement of the corollary, an

---

<sup>2</sup>For reasons irrelevant to ordinary mathematics, prime ('') has a special meaning in TLA<sup>+</sup>.

ordinary prose assertion is interpreted as an ASSUME/PROVE when it is the statement of the theorem to be proved. If it appears as a proof step, we can try indicating the ASSUME/PROVE structure by writing “If we assume …, then we can prove …”. That and the hierarchical structuring may be good enough to convey the intended meaning. However, it seems safer to introduce ASSUME and PROVE as keywords and explain their meaning to the reader. Fortunately, ASSUME/PROVE steps are not common in informal proofs.

### SUFFICES

Step 1 in Figure 3 can be written in TLA<sup>+</sup> as this SUFFICES step:

```
SUFFICES ASSUME NEW $a \in I$, NEW $b \in I$, $a < b$
 PROVE $f[b] > f[a]$
```

This step asserts the truth of the formula

$$(2) \quad (\forall a, b \in I : (a < b) \Rightarrow (f[b] > f[a])) \Rightarrow \text{IsIncreasingOn}(f, I)$$

where the hypothesis of the implication (2) is the assertion of the ASSUME/PROVE, and the conclusion *IsIncreasingOn*( $f$ ,  $I$ ) of (2) is the proof’s current goal. The step changes the current goal of the proof to be the PROVE formula  $f[b] > f[a]$ ; and it allows the assumptions  $a \in I$ ,  $b \in I$ , and  $a < b$  of the ASSUME clause to be assumed in the rest of the proof. These assumptions do not apply to the proof of the step itself, and the NEW variables  $a$  and  $b$  are meaningless in that proof.

A proof by contradiction of a simple assertion  $F$  begins with the step

```
SUFFICES ASSUME $\neg F$
 PROVE FALSE
```

A SUFFICES step can also have the form SUFFICES  $F$  for a formula  $F$ . Since a formula is an ASSUME/PROVE with no assumptions, this step asserts that  $F$  implies the proof’s current goal, and the step’s proof must prove this assertion. The step changes the current goal to  $F$ .

The SUFFICES construct is not necessary; any SUFFICES can be eliminated by restructuring the proof. For example, we could eliminate the “It suffices to” from the proof of Figure 3 by using the following top-level structure:

1. ASSUME 1.  $a$  and  $b$  are points in  $I$ .
  2.  $a < b$
- ```
PROVE  $f[b] > f[a]$ 
```

2. Q.E.D.

PROOF: By 1 and definition of an increasing function.

The proof of the original step 1 appears in the proof of the new Q.E.D step; steps 2–5 of the original proof become the proof of the new step 1.

As this example illustrates, removing a SUFFICES adds one level to the proof. Proofs are generally easiest to read if each level contains about 4 to 10 steps. The SUFFICES step eliminates the kind of two-step proof that would otherwise be needed to prove the corollary.

Step 1 of Figure 3 shows that a SUFFICES is easily expressed with informal prose, even if it is a SUFFICES ASSUME/PROVE.

PICK

If we were to expand the proof of step 4 of Figure 3, it would look like this.

4.1. Pick x in (a, b) with $f'(x) = \frac{f(b) - f(a)}{b - a}$.

PROOF: Such an x exists by step 2.

4.2. Q.E.D.

PROOF: By 4.1 and 3.

Step 4.1 is expressed formally by a step of the form

PICK $x \in S : P(x)$

This step introduces the new variable x and asserts that $P(x)$ is true. The step's proof must show that there exists an x satisfying $P(x)$.

There is no problem expressing PICK in prose.

CASE

If F is a formula, the step CASE F is an abbreviation for

ASSUME F

PROVE Q.E.D.

where Q.E.D. stands for the formula that is the current goal. An ordinary proof by cases ends with a sequence of CASE steps followed by the Q.E.D. step. Here is a typical example:

1. PICK $n \in S : \dots$
2. CASE $n \geq 0$
3. CASE $n < 0$

4. Q.E.D.

PROOF: By 1, 2, and 3.

In general, the Q.E.D. step's proof cites the CASE statements and shows that the cases are exhaustive—which in this example is presumed to be trivial because S is a set of numbers.

It is easy to express a CASE statement in prose—for example:

We now consider the case in which $n \geq 0$.

However, why not just write CASE $n \geq 0$? Readers will understand what it means.

Definitions

It is often convenient to give some expression a name in part of the proof. TLA⁺ allows definitions as proof steps. A single step can contain multiple definitions. The definition is in effect for the rest of the proof's current level.

4.2 Hierarchical Numbering

Figure 4 introduced a naming scheme in which, for example, 3.3.4 is the 4th step in the proof of step 3.3. The most obvious problem with this scheme is that step names get longer as the proof gets deeper. For steps beyond level 4, the names are too hard to read and take up too much space.

A less obvious problem is that any step can be mentioned anywhere in the proof. One can refer to step 2.4.1 from inside the proof of step 3.3.4. Such a reference should not be allowed because it violates the hierarchical structuring of the proof. It is illegal to use step 2.4.1 in the proof of step 3.3.4 if step 2 or step 2.4 is an ASSUME/PROVE, because step 2.4.1 has then been proved under assumptions that may not hold for the proof of 3.3.4. Even if there is no ASSUME/PROVE making the reference illegal, such a reference makes the proof harder to read. The proof of step 3.3.4 should refer only to the following steps: 1, 2, 3.1, 3.2, 3.3.1, 3.3.2, and 3.3.3.

In TLA⁺, the 4th step of a level 3 proof is named $\langle 3 \rangle 4$. A proof can have many different steps named $\langle 3 \rangle 4$. However, at any point in the proof, only one of them can be referred to without violating the hierarchical structure. (Proving this is a nice little exercise.) Any valid reference to step $\langle 3 \rangle 4$ refers to the most recent preceding step with that name. This numbering scheme is used in the TLA⁺ proof of Spivak's corollary in the appendix.

PROOF: $\Pi = \mathcal{C}(\Pi \wedge L_1)$ [Hypothesis 1]
 $\subseteq \mathcal{C}(\Pi \wedge L_2)$ [Hypothesis 2 and monotonicity of closure]
 $\subseteq \mathcal{C}(\Pi)$ [monotonicity of closure]
 $= \Pi$ [Hypothesis 1, which implies Π closed]

This proves that $\Pi = \mathcal{C}(\Pi \wedge L_2)$.

Figure 5: An example of equational reasoning.

4.3 Equational Proofs

A different kind of structuring is provided by equational reasoning. An equational proof consists of a sequence of relations and their proofs, from which one deduces a relation by transitivity—for example, we prove

$$(3) \quad a + 1 \leq b^2 - 3 = \sqrt{c - 42} \leq d$$

and deduce $a + 1 \leq d$. I find such proofs to be elegant, and I use them when I can. Figure 5 shows a more sophisticated type of equational reasoning using set inclusion; it is a slightly modified version of a proof from a paper of which I was an author [1].

Equational proofs have a serious drawback when printed on paper: there seems to be no good way to display hierarchical proofs of the individual relations. On paper, equational reasoning works well only as a lowest-level proof, where the proof of each relation is very short—as in Figure 5. TLA⁺ provides the following way of writing the sequence of relations (3) in a proof:

$$\begin{aligned} \langle 1 \rangle 1. \quad & a + 1 \leq b^2 - 3 \\ \langle 1 \rangle 2. \quad & @ = \sqrt{c - 42} \\ \langle 1 \rangle 3. \quad & @ \leq d \end{aligned}$$

where the @ symbol stands for the preceding expression. However, this lacks the visual simplicity of (3). There is no problem displaying hierarchically structured equational proofs with hypertext.

4.4 Comments

Like any computer-readable language, TLA⁺ allows comments to aid human readers. With hypertext, comments can be attached to any part of a proof, to be popped up and hidden as the reader wishes. On paper, comments in arbitrary places can be distracting. However, there is one form of comment that works well on paper for both formal and informal proofs: a proof sketch that comes between a statement and its proof. The sketch can explain the

intuition behind the proof and can point out the key steps. Proof sketches can be used at any level in a hierarchical proof, including before the highest-level proof. A reader not interested in the details of that part of the proof can read just the proof sketch and skip the steps and their proofs.

4.5 Completely Formal Proofs

In principle, the proof of a theorem should show that the theorem can be formally deduced from axioms by the application of proof rules. In practice, we never carry a proof down to that level of detail. However, a mathematician should always be able to keep answering the question *why?* about a proof, all the way down to the level of axioms. A completely formal proof is the Platonic ideal.

Most mathematicians have no idea how easy it is to formalize mathematics. Their image of formalism is the incomprehensible sequences of symbols found in *Principia Mathematica*. The appendix contains formal TLA⁺ definitions of intervals, limits, continuity, and the derivative, assuming only the definitions of the real numbers and of ordinary arithmetic operations. I expect most readers will be surprised to learn that this takes only 19 lines. The appendix also contains a TLA⁺ proof of Spivak's corollary that has been checked by the TLAPS proof system [6].

Formalizing mathematics is easy, but writing formal, machine-checkable proofs is not. It will be decades before mechanical proof checkers are good enough that writing a machine-checked proof is no harder than writing a careful informal proof. Until then, there is little reason for a mathematician to write formal mathematics.

However, there is good reason for teaching how to write a formal proof as part of a standard mathematics education. Mathematicians think that the logic of the proofs they write is completely obvious, but our examination of Spivak's proof shows that they are wrong. Students are expected to learn how to write logically correct proofs from examples that, when read literally, are illogical. (Recall the first sentence of Spivak's proof.) It is little wonder that so few of them succeed. Learning to write structured formal proofs that a computer can check will teach students what a proof is. Going from these formal proofs to structured informal proofs would then be a natural step.

Is it crazy to think that students who cannot learn to write proofs in prose can learn to write them in an unfamiliar formal language and get a computer to check them? Anyone who finds it crazy should consider how many students learn to write programs in unfamiliar formal languages and

get a computer to execute them, and how few now learn to write proofs.

For reasons mentioned in the appendix, TLA⁺ and its TLAPS prover are not ideal for teaching mathematics students. However, the structured proofs of TLA⁺ make it the best currently available language for the task that I know of. It should be satisfactory for writing proofs in some particular domain such as elementary group theory.

5 Experience

I am a computer scientist who was educated as a mathematician. I discovered structured proofs through my work on concurrent (multiprocess) algorithms. These algorithms can be quite subtle and hard to get right; their correctness proofs require a degree of precision and rigor unknown to most mathematicians (and many computer scientists). A missing hypothesis, such as that a set must be nonempty, which is a trivial omission in a mathematical theorem, can mean a serious bug in an algorithm.

Proofs of algorithms are most often mathematically shallow but complicated, requiring many details to be checked. With traditional prose proofs, I found it impossible to make sure that I had not simply forgotten to check some detail. Computer science offers a standard way to handle complexity: hierarchical structure. Structured proofs were therefore an obvious solution. They worked so well for proofs of algorithms that I tried them on the more mathematical proofs that I write. I have used them for almost every proof of more than about ten lines that I have published since 1991. (The only exceptions I can find are in a paper in which the proofs served only to illustrate how certain formal proof rules are used.)

My earlier paper on structured proofs described how effective they are at catching errors. It recounted how only by writing such a proof was I able to re-discover an error in a proof of the Schroeder-Bernstein theorem in a well-known topology text [2, page 28]. I recently received email from a mathematician saying that he had tried unsuccessfully to find that error by writing a structured proof. I asked him to send me his proof, and he responded:

I tried typing up the proof that I'd hand-written, and in the process, I think I've found the fundamental error... I now really begin to understand what you mean about the power of this method, even if it did take me hours to get to this point!

It is instructive that, to find the error, he had to re-write his proof to be

read by someone else. Eliminating errors requires care. Structured proofs make it possible, not inevitable.

Over the years, I have published quite a few papers with structured proofs. For proofs with the level of detail typical of mathematics papers, I cannot remember any reader commenting on the proof style unless explicitly asked to. Structured proofs are easy enough to read that one forgets about the form and concentrates on the content. However, I have also published some papers with long, excruciatingly detailed proofs. I am reluctant to publish a paper with a short proof if I would not have been able to find the correct result without writing a longer one. Here are a referee's comments on one such proof.

The proofs... are lengthy, and are presented in a style which I find very tedious. I think the readers... are going to be more interested in understanding the techniques and how they can apply them, than they will be in reading the formal proofs. A problem with the proofs is that they do not clearly distinguish the trivial manipulations from the nontrivial or surprising parts. ... My feeling is that informal proof sketches... to explain the crucial ideas in each result would be more appropriate.

I think the referee would have found the proofs much more tedious if written in a conventional prose style, but my co-author and I could have made the proof easier to read had we used the proof sketches described in Section 4.4 above. I do agree that the proofs were too long and detailed for the journal's readers. Today, the obvious approach would be to put the long proof on the Web and publish a proof sketch with a link to the real proof. The Web was in its infancy when the paper was published, but the editor agreed to publish the proofs as a separate appendix available only on-line. None of the first reviewers had read the proofs, so the editor found another referee to do that. When asked how he or she found the proof style, the referee responded:

I have found the hierarchical structuring of proofs to be very helpful, if read top-down according to the suggestions of the authors. In fact, it might well be the only way to present long proofs... in a way that is both detailed (to ensure correctness) and readable. For long proofs, I think that describing the idea of the proof in a few words at the beginning (if appropriate) would help make them more understandable. ... But in general, I found the structured approach very effective.

6 Objections to Structured Proof

When lecturing about structured proofs, I have heard many objections to them. I cannot recall any objection that I found to be based on a rational argument; they have all been essentially emotional. Here are three common ones.

They are too complicated.

In lectures, I usually flash on the screen one of my multi-page structured proofs. People have reacted by saying that the structuring makes the proof too complicated, as if replacing the numbering and indentation by prose would magically simplify the proof. One mathematician described how she had explained a beautiful little proof by Hardy to an audience of non-mathematicians, and that the audience could not possibly understand my proofs. She apparently believed that structuring Hardy's tiny proof would turn it into multiple pages full of obscure symbols.

It is an unfortunate fact that being rigorous requires filling in missing details, which makes a proof longer. As we saw with Spivak's corollary, a structured proof makes it easier to see what is missing; this would lead mathematicians to correct the omissions, resulting in longer proofs. Fortunately, structuring allows us to add those details without making the proof any harder to read as hypertext, and only a little harder to read on paper.

They don't explain why the proof works.

Mathematicians seem to think that their proofs explain themselves. I cannot remember reading a mathematician's proof that was both a proof and an explanation of why the proof works. It's hard enough to make the structure of a prose proof clear; doing it while also providing an intuitive explanation is a formidable task.

I suspect that this objection is based on confusing a proof sketch with a proof. Proof sketches are fine, but they are not proofs. Mathematicians sometimes precede a proof with a proof sketch, but there is no easy way to relate the steps of the proof with the proof sketch. The ability to add proof sketches at any level of a structured proof makes it possible to provide a much clearer explanation of why a proof works.

A proof should be great literature.

This is nonsense. A proof should not be great literature; it should be beautiful mathematics. Its beauty lies in its logical structure, not in its prose. Proofs are more like architecture than like literature, and architects do not use prose to design buildings. Prose cannot add to the beauty of $e^{i\pi} + 1 = 0$, and it is a poor medium for expressing the beauty of a proof.

7 Beginning

When I started writing structured proofs, I quickly found them to be completely natural. Writing non-trivial prose proofs now seems as archaic to me as writing

The number e raised to the power of i times π , when added to 1, equals 0.

Imagine how many errors we would make performing algebraic calculations with equations written in prose. Writing proofs in prose is equally error prone. As I reported in my earlier paper, anecdotal evidence indicates that a significant fraction of published mathematics contains serious errors. This will not change until mathematicians understand that precision and rigor, not prose, are the natural medium of mathematics, and they stop writing 17th century proofs.

Fortunately, it's not hard to write 21st century proofs. There is no need to wait until other mathematicians are doing it. You can begin by just adding structure to an existing proof, as we did with Spivak's proof. Start by rewriting a simple proof and then try longer ones. You should soon find this a much more logical way to write your proofs, and readers will have no trouble understanding the proof style.

Writing structured proofs is liberating. It allows you to concentrate on logical structure instead of sentence structure. You will no longer waste your time searching for different ways to say *therefore*. To help you typeset your structured proofs, a L^AT_EX package and an associated computer program are available on the Web [4].

Acknowledgement

In his courses on analysis and algebraic topology, Richard Palais taught me how mathematics could be made precise and rigorous, and thereby more beautiful.

References

- [1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
- [2] John L. Kelley. *General Topology*. The University Series in Higher Mathematics. D. Van Nostrand Company, Princeton, New Jersey, 1955.
- [3] Leslie Lamport. TLA—temporal logic of actions. A web page, a link to which can be found at URL <http://lamport.org>. The page can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.
- [4] Leslie Lamport. Useful LaTeX packages. <http://research.microsoft.com/en-us/um/people/lamport/latex/latex.html>. The page can also be found by searching the Web for the 23-letter string formed by concatenating `uid` and `lamportlatexpackages`.
- [5] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., 1993. A symposium in honor of Richard Palais’ sixtieth birthday. Also published in *American Mathematical Monthly*, 102(7):600–608, August–September 1995.
- [6] Microsoft Research-INRIA Joint Centre. Tools and methodologies for formal specifications and for proofs. <http://www.msr-inria.inria.fr/Projects/tools-for-formal-specs>.
- [7] Michael Spivak. *Calculus*. W. A. Benjamin, Inc., New York, 1967.

Appendix: A Formal Proof

This appendix contains a TLA⁺ formalization of Spivak's proof, preceded by formal definitions of the necessary concepts of differential calculus and statements of the two theorems used in that proof. This is all done in a TLA⁺ module named *Calculus*.

Mathematicians will be struck by the absence of some common mathematical notation. For example, the open interval (a, b) is written *OpenInterval*(a, b). No single syntax can capture the wide variety of notation used in mathematics, where (a, b) may be an interval or an ordered pair, depending on the context. A good formal language for math should permit such context-dependent notation. TLA⁺ was not designed for use by mathematicians, so it provides conventional notation mainly for basic operators of logic and set theory.

TLAPS uses sophisticated algorithms for performing simple reasoning about integers. However, because it is still under development and real numbers are seldom used in algorithms, TLAPS does not yet provide any special support for reasoning about reals. The proof of Spivak's corollary therefore assumes without proof five very simple facts about real numbers.

The appendix shows the typeset version of the *Calculus* module, not its actual ASCII text. For example, the definition of *OpenInterval* that is typeset as

$$\text{OpenInterval}(a, b) \triangleq \{r \in \text{Real} : (a < r) \wedge (r < b)\}$$

appears in the module as

```
OpenInterval(a, b) == {r \in Real : (a < r) /\ (r < b)}
```

The L^AT_EX source for the typeset version was generated from the ASCII by a program, but it is possible that editing of the document introduced errors.

The rest of the appendix consists of the *Calculus* module together with interspersed comments explaining some of the TLA⁺ notation. These explanations and a thorough knowledge of calculus should allow you to figure out what's going on. However, you probably won't find it easy reading. This is hardly surprising, since the definition of the derivative, which is line 20 of the module, appears on page 127 of Spivak's book. With a few pages of explanation, I expect a mathematician would find the TLA⁺ formulas as easy to read as Spivak's text.

The following EXTENDS statement imports the standard *Reals* module that defines the set *Real* of real numbers and the usual operators on real numbers such as / (division) and \leq .

EXTENDS *Reals*

$$\text{OpenInterval}(a, b) \triangleq \{r \in \text{Real} : (a < r) \wedge (r < b)\}$$

$$\text{ClosedInterval}(a, b) \triangleq \{r \in \text{Real} : (a \leq r) \wedge (r \leq b)\}$$

In TLA⁺, SUBSET *S* is the power set (the set of all subsets) of *S*.

$$\text{SetOfIntervals} \triangleq \{S \in \text{SUBSET Real} : \forall x, y \in S : \text{OpenInterval}(x, y) \subseteq S\}$$

UNION *S* is the union of all sets that are elements of the set *S*, and $[S \rightarrow T]$ is the set of functions with domain *S* and range a subset of *T*. Hence, the following defines *RealFunction* to be the set of all real-valued functions whose domain is a set of real numbers.

$$\text{RealFunction} \triangleq \text{UNION } \{[S \rightarrow \text{Real}] : S \in \text{SUBSET Real}\}$$

$$\text{AbsoluteValue}(a) \triangleq \text{IF } a > 0 \text{ THEN } a \text{ ELSE } -a$$

$$\text{OpenBall}(a, e) \triangleq \{x \in \text{Real} : e > \text{AbsoluteValue}(x - a)\}$$

$$\text{PositiveReal} \triangleq \{r \in \text{Real} : r > 0\}$$

TLA⁺ uses square brackets for function application, as in $f[x]$. Parentheses are reserved for arguments of defined operators like *OpenInterval*.

$$\begin{aligned} \text{IsLimitAt}(f, a, b) &\triangleq (b \in \text{Real}) \wedge \\ &\forall e \in \text{PositiveReal} : \exists d \in \text{PositiveReal} : \\ &\quad \forall x \in \text{OpenBall}(a, d) \setminus \{a\} : f[x] \in \text{OpenBall}(b, e) \end{aligned}$$

$$\text{IsContinuousAt}(f, a) \triangleq \text{IsLimitAt}(f, a, f[a])$$

$$\text{IsContinuousOn}(f, S) \triangleq \forall x \in S : \text{IsContinuousAt}(f, x)$$

Mathematics provides no formal notation for writing a function. In TLA⁺, $[x \in S \mapsto e(x)]$ is the function with domain *S* that maps *x* to *e(x)* for every *x* in *S*. If *f* is a function, then DOMAIN *f* is its domain.

$$\begin{aligned} \text{IsDerivativeAt}(f, a, b) &\triangleq \\ &\exists e \in \text{PositiveReal} : \\ &\quad (\text{OpenBall}(a, e) \subseteq \text{DOMAIN } f) \wedge \\ &\quad \text{IsLimitAt}([x \in \text{OpenBall}(a, e) \setminus \{a\} \mapsto (f[x] - f[a])/(x - a)], a, b) \end{aligned}$$

$$\text{IsDifferentiableAt}(f, a) \triangleq \exists b \in \text{Real} : \text{IsDerivativeAt}(f, a, b)$$

$$\text{IsDifferentiableOn}(f, S) \triangleq \forall x \in S : \text{IsDifferentiableAt}(f, x)$$

CHOOSE $x : P(x)$ is an arbitrary value x satisfying $P(x)$, if such a value exists; otherwise its value is unspecified. The CHOOSE operator is known to logicians as Hilbert's ε .

$$d(f) \triangleq [x \in \text{DOMAIN } f \mapsto \text{CHOOSE } y : \text{IsDerivativeAt}(f, x, y)]$$

The following THEOREM asserts the truth of the formula $\forall f : \dots$ and names that formula *Theorem1*.

$$\begin{aligned} \text{THEOREM } \text{Theorem1} &\triangleq \forall f \in \text{RealFunction} : \forall a \in \text{Real} : \\ &\quad \text{IsDifferentiableAt}(f, a) \Rightarrow \text{IsContinuousAt}(f, a) \end{aligned}$$

$$\text{IsIncreasingOn}(f, I) \triangleq \forall x, y \in I : (x < y) \Rightarrow (f[x] < f[y])$$

$$\begin{aligned} \text{THEOREM } \text{MeanValueTheorem} &\triangleq \\ \forall f \in \text{RealFunction} : \forall a, b \in \text{Real} : & \\ (& (a < b) \\ & \wedge \text{IsContinuousOn}(f, \text{ClosedInterval}(a, b)) \\ & \wedge \text{IsDifferentiableOn}(f, \text{OpenInterval}(a, b))) \\ \Rightarrow & (\exists x \in \text{OpenInterval}(a, b) : \\ & d(f)[x] = (f[b] - f[a])/(b - a)) \end{aligned}$$

We assume without proof the following five trivial facts about real numbers. TLAPS easily proves the first four for integers.

$$\text{PROPOSITION } \text{Fact1} \triangleq \forall x \in \text{Real} : x - x = 0$$

$$\text{PROPOSITION } \text{Fact2} \triangleq \forall x, y \in \text{Real} : (x \leq y) \equiv (x < y) \vee (x = y)$$

$$\text{PROPOSITION } \text{Fact3} \triangleq \forall x, y \in \text{Real} : x - y \in \text{Real}$$

$$\text{PROPOSITION } \text{Fact4} \triangleq \forall x, y \in \text{Real} : (x < y) \equiv (y - x > 0)$$

$$\text{PROPOSITION } \text{Fact5} \triangleq \forall x, y \in \text{Real} : (y > 0) \wedge (x/y > 0) \Rightarrow (x > 0)$$

Below is the corollary and its proof. Each step of the high-level proof formalizes the correspondingly-numbered step of the proof in Figure 3, where step $\langle 1 \rangle 1a$ has been added in the formal proof. The proof of step $\langle 1 \rangle 2$ formalizes the proof of Figure 4.

The lowest-level paragraphs of an informal proof are replaced with BY statements that say what facts and definitions (DEF) are used. (Assumptions in NEW declarations and in the statement of the corollary are automatically used by TLAPS.) The proof has been decomposed for the benefit of TLAPS, not for a human reader.

$$\begin{aligned} \text{COROLLARY } \text{Spivak} &\triangleq \text{ASSUME NEW } f \in \text{RealFunction}, \\ &\quad \text{NEW } I \in \text{SetOfIntervals}, \\ &\quad \text{IsDifferentiableOn}(f, I), \\ &\quad \forall x \in I : d(f)[x] > 0 \\ \text{PROVE } &\text{IsIncreasingOn}(f, I) \end{aligned}$$

$\langle 1 \rangle 1.$ SUFFICES ASSUME NEW $a \in I$, NEW $b \in I$, $a < b$
 PROVE $f[a] < f[b]$
 BY DEF *IsIncreasingOn*

$\langle 1 \rangle 1a.$ $(OpenInterval(a, b) \subseteq I)$
 $\wedge (ClosedInterval(a, b) \subseteq I)$
 $\wedge (OpenInterval(a, b) \subseteq ClosedInterval(a, b))$
 $\langle 2 \rangle 1.$ $OpenInterval(a, b) \subseteq I$
 BY DEF *SetOfIntervals*
 $\langle 2 \rangle 2.$ $ClosedInterval(a, b) = OpenInterval(a, b) \cup \{a, b\}$
 BY $\langle 1 \rangle 1$, *Fact2* DEF *ClosedInterval*, *OpenInterval*, *SetOfIntervals*
 $\langle 2 \rangle 3.$ QED
 BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$

$\langle 1 \rangle 2.$ $\exists x \in OpenInterval(a, b) : d(f)[x] = (f[b] - f[a])/(b - a)$
 $\langle 2 \rangle 1.$ *IsDifferentiableOn*(f , $ClosedInterval(a, b)$)
 BY $\langle 1 \rangle 1a$, $\langle 1 \rangle 1$ DEF *IsDifferentiableOn*
 $\langle 2 \rangle 2.$ *IsContinuousOn*(f , $ClosedInterval(a, b)$)
 $\langle 3 \rangle 1.$ SUFFICES ASSUME NEW $x \in ClosedInterval(a, b)$
 PROVE *IsContinuousAt*(f , x)
 BY DEF *IsContinuousOn*
 $\langle 3 \rangle 2.$ *IsDifferentiableAt*(f , x)
 BY $\langle 2 \rangle 1$ DEF *IsDifferentiableOn*
 $\langle 3 \rangle 3.$ QED
 BY $\langle 3 \rangle 2$, *Theorem1* DEF *ClosedInterval*
 $\langle 2 \rangle 3.$ QED
 BY $\langle 1 \rangle 1$, $\langle 1 \rangle 1a$, $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, *MeanValueTheorem*
 DEF *SetOfIntervals*, *IsDifferentiableOn*

$\langle 1 \rangle 3.$ $\forall x \in OpenInterval(a, b) : d(f)[x] > 0$
 BY $\langle 1 \rangle 1a$

$\langle 1 \rangle 4.$ $(f[b] - f[a])/(b - a) > 0$
 $\langle 2 \rangle 1.$ PICK $x \in OpenInterval(a, b) :$
 $d(f)[x] = (f[b] - f[a])/(b - a)$
 BY $\langle 1 \rangle 2$
 $\langle 2 \rangle 2.$ $d(f)[x] > 0$
 BY $\langle 2 \rangle 1$ DEF *SetOfIntervals*
 $\langle 2 \rangle 3.$ QED
 BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$

⟨1⟩5. QED
 ⟨2⟩1. $(a \in \text{Real}) \wedge (b \in \text{Real})$
 BY DEF *SetOfIntervals*
 ⟨2⟩2. $(f[a] \in \text{Real}) \wedge (f[b] \in \text{Real})$
 ⟨3⟩1. SUFFICES ASSUME NEW $x \in \text{Real}$,
 IsDifferentiableAt(f , x)
 PROVE $f[x] \in \text{Real}$
 BY ⟨2⟩1 DEF *IsDifferentiableOn*
 ⟨3⟩2. $\forall e \in \text{PositiveReal} : x \in \text{OpenBall}(x, e)$
 BY *Fact1* DEF *OpenBall*, *PositiveReal*, *AbsoluteValue*
 ⟨3⟩3. $x \in \text{DOMAIN } f$
 BY ⟨3⟩1, ⟨3⟩2 DEF *IsDifferentiableAt*, *IsDerivativeAt*
 ⟨3⟩4. QED
 BY ⟨3⟩3 DEF *IsDifferentiableAt*, *IsDerivativeAt*, *RealFunction*
 ⟨2⟩3. $(f[b] - f[a] \in \text{Real}) \wedge (b - a \in \text{Real})$
 BY ⟨2⟩1, ⟨2⟩2, *Fact3*
 ⟨2⟩4. $f[b] - f[a] > 0$
 BY ⟨1⟩1, ⟨1⟩4, ⟨2⟩1, ⟨2⟩3, *Fact4*, *Fact5*
 ⟨2⟩5. QED
 BY ⟨2⟩2, ⟨2⟩4, *Fact4*