

**Zohar Manna**  
**Amir Pnueli**

# **Temporal Verification**

---

# **Springer**

*New York*

*Berlin*

*Heidelberg*

*Barcelona*

*Budapest*

*Hong Kong*

*London*

*Milan*

*Paris*

*Tokyo*

*Zohar Manna* . . . *Amir Pnueli*

# Temporal Verification of Reactive Systems

*Safety*

With 181 Illustrations



Springer

Zohar Manna  
Department of Computer Science  
Stanford University  
Stanford, CA 94305 USA

Amir Pnueli  
Computer Science Department  
Weizmann Institute of Science  
Rehovot, 76100 Israel

Library of Congress Cataloging-in-Publication Data

Manna, Zohar.

Temporal verification of reactive systems:safety / Zohar Manna  
and Amir Pnueli.

p. cm.

Includes bibliographical references and index.

ISBN 0-387-94459-1

1. Computer software—Verification. 2.

Parallel processing

(Electronic computers) I. Pnueli, A. II. Title.

005.2—dc20

95-5442

Printed on acid-free paper.

© 1995 Springer-Verlag New York, Inc.

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA) except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Production managed by Hal Henglein; manufacturing supervised by Joe Quatela.

Photocomposed copy prepared from the authors' TeX file.

Printed and bound by R.R. Donnelley and Sons, Harrisonburg, VA.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-94459-1 Springer-Verlag New York Berlin Heidelberg

**To our families:**

<i>Nitza</i>	<i>Ariela</i>
<i>Yonit</i>	<i>Shira</i>
<i>Hagit</i>	<i>Ishay</i>
<i>Irit</i>	
<i>Amit</i>	<i>Noga</i>



# Preface

This book is about the verification of reactive systems. A *reactive system* is a system that maintains an ongoing interaction with its environment, as opposed to computing some final value on termination. The family of reactive systems includes many classes of programs whose correct and reliable construction is considered to be particularly challenging, including concurrent programs, embedded and process control programs, and operating systems. Typical examples of such systems are an air traffic control system, programs controlling mechanical devices such as a train, or perpetually ongoing processes such as a nuclear reactor.

With the expanding use of computers in safety-critical areas, where failure is potentially disastrous, correctness is crucial. This has led to the introduction of *formal verification techniques*, which give both users and designers of software and hardware systems greater confidence that the systems they build meet the desired specifications.

## Framework

The approach promoted in this book is based on the use of *temporal logic* for specifying properties of reactive systems, and develops an extensive verification methodology for proving that a system meets its temporal specification. Reactive programs must be specified in terms of their ongoing behavior, and temporal logic provides an expressive and natural language for specifying this behavior. Our framework for specifying and verifying temporal properties of reactive systems is based on the following four components:

1. A *computational model* to describe the behavior of reactive systems. The model adopted in this book is that of a *Fair Transition System* (FTS).
2. A *system description language* to express proposed implementations. Here we use a *Simple Programming Language* (SPL) for this purpose.
3. A *specification language* to express properties that should be satisfied by any proposed implementation. The language of choice in this text is *Temporal Logic* (TL).

4. *Verification techniques* to prove that the proposed implementation meets its prescribed specification. We consider two types of verification techniques. The first is a *deductive approach*, which provides a set of verification rules that reduce the task of proving a temporal property to checking the validity of first-order formulas. The second is an *algorithmic approach* (“model checking”) which presents algorithms for automatic verification of temporal properties.

Figure A illustrates how the four parts of the framework interact.

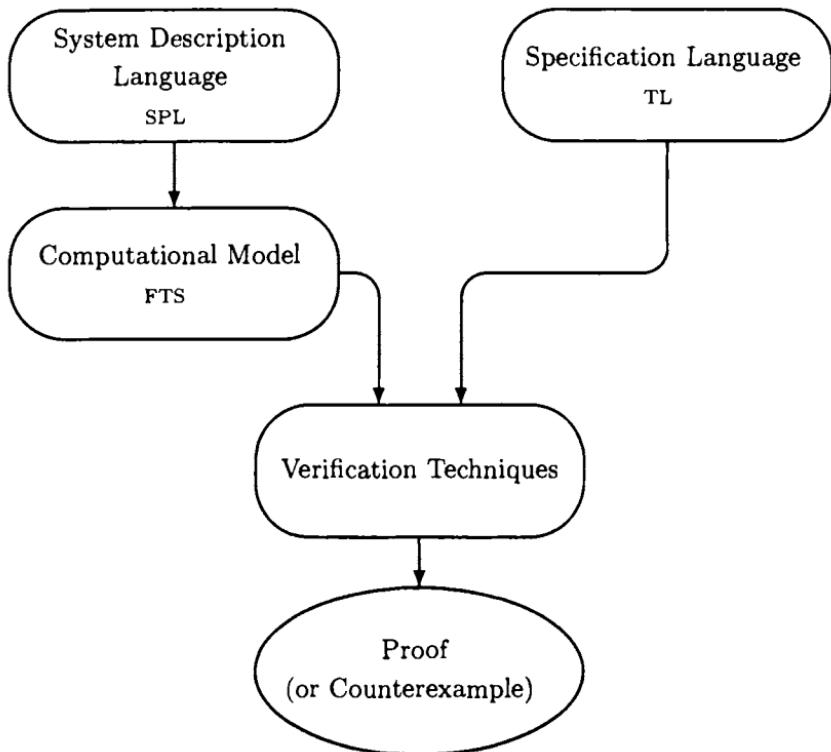


Fig. A. The four parts of the framework.

Temporal properties of systems can be classified according to the form of the formulas expressing them. A *safety property* is a property that can be specified by a safety formula of the form  $\Box p$  (using the temporal operator  $\Box$  meaning *always*). This formula states that the property  $p$  holds throughout the computation.

Consider, for example, a system that controls the gates to a railroad crossing. The requirement “whenever the train is passing, the gates should be down”

is a property expressible by a formula  $\square p$  where  $p$  characterizes all states except those in which the train is passing while the gate is up. Typically, safety properties specify that nothing “bad” ever happens, such as allowing a train in the intersection while the gate is up.

However, there are additional properties that cannot be specified by such safety formulas. For example, one such property expresses the requirement “if the intersection has been clear of trains for a sufficiently long time, then eventually a train will arrive or the gate will be raised.” Properties that cannot be specified by a safety formula  $\square p$  are called *progress properties*. Typically, progress properties specify that something “good” eventually happens, such as the gate eventually being raised after a long absence of trains. Progress properties are further classified into several distinct classes according to the formulas expressing them.

Our verification methodology partitions the verification rules according to this classification, presenting rules for each class that are specific and effective for proving properties belonging to this class.

## The Three Volumes

This book is the second in a sequence of three volumes (published by Springer-Verlag) organized as follows:

- Volume I (appeared in 1991): “The Temporal Logic of Reactive and Concurrent Systems: Specification.” This volume studies in great detail the computational model, the system description language, the language of temporal logic and its use for specifying system properties. We refer to Volume I as the **SPEC** book.
- Volume II (This volume): “Temporal Verification of Reactive Systems: Safety.” This volume presents verification techniques for proving safety properties. We refer to this volume as the **SAFETY** book.
- Volume III: “Temporal Verification of Reactive Systems: Progress.” This volume presents verification techniques for proving progress properties. We refer to Volume III as the **PROGRESS** book.

Even though this is the second volume in the sequence, it is largely self-contained and can be studied or taught independently of the first volume. In particular, Chapter 0 includes all the material from the first volume that is relevant to temporal verification. Naturally, most of the discussions and examples have been omitted, and we encourage the more inquisitive reader to consult the **SPEC** book whenever a more detailed explanation or motivation is needed.

## Verification Techniques

As previously explained, this volume is devoted to the verification of safety properties. While all safety properties are specifiable by a formula of the form  $\square p$ , it

is possible to distinguish several subclasses of the safety class according to more specialized formulas expressing them.

The main approach to verification studied in this volume (Chapters 1-4) is that of *deductive verification*: program properties are verified using formal deduction based on a set of inference rules. Such rules are presented first for some of the simpler subclasses, followed by rules for the entire safety class. A convenient and visual representation of the structure of a proof is provided by *verification diagrams*. Each set of rules, corresponding to a subclass or to the entire safety class, is shown to be complete for proving properties belonging to that class. This ensures that

Every correct property of a system can be formally proven,  
using the deductive techniques presented in this book.

In our presentation, we attempt to emphasize the feasibility, practicality, and universality of the deductive approach.

In addition to the deductive approach, we also consider a second verification technique: *algorithmic verification* (often referred to as “model checking”). This technique is based on algorithms that, without any user guidance or ingenuity, check automatically whether a given system satisfies its temporal specification. The obvious advantages of this approach are offset by the fact that it usually can only be applied to *finite-state systems*, i.e., systems restricted to having only finitely many distinct states. In this volume we present several algorithms for checking that a given temporal formula is valid over a given finite-state system. In some cases these algorithms are also applicable to systems with infinitely many states. The presentation of algorithmic verification is also organized by subclasses of safety properties; we provide an algorithm for each subclass.

In addition to these class-specific algorithms, we provide in Chapter 5 a universal algorithm for checking whether an arbitrary temporal formula is valid over a given finite-state system.

## Intended Audience

This book is intended for people who are interested in the analysis of reactive systems and wish to find out how formal techniques can be used to ensure their accuracy and correctness.

We assume our reader has some familiarity with programming and programming languages. In particular, some acquaintance with basic concurrency concepts is desirable, e.g., use of semaphores, deadlock, and live-lock; however, no prior knowledge of a concurrent language is necessary. We also assume a reasonable understanding of first-order logic and the notions of validity and provability by deductive systems. However, no knowledge of temporal logic is needed, since this topic is introduced in Chapter 0.

## Contents

Chapter 0 presents the basic building blocks of the temporal framework. The chapter begins by introducing the fair transition system as a computational model for representing reactive systems. We then introduce the syntax and semantics of a simple programming language SPL used as our system description language. The SPL language contains programming constructs allowing for concurrency and communication through both shared variables and message passing. We also present the syntax and semantics of our specification language, temporal logic.

In Chapters 1 and 2, we consider the special case of properties specifiable by a formula  $\Box p$ , where  $p$  is a first-order formula (also called a state formula or an assertion). We refer to such properties as *state invariances* and say that  $p$  is an *invariant* of the considered program. Chapter 1 presents the basic methods for proving such properties. A major task in the verification of a safety property is the identification of an *inductive assertion*, i.e., an assertion which implies the property of interest and can be proved by induction. Chapter 1 addresses this task by providing heuristics for the construction of such inductive assertions. Another useful technique introduced in the chapter is the derivation of simple invariants according to the syntactic structure of the program. In many cases, this derivation can be done automatically and significantly simplifies the task of verification.

Chapter 2 illustrates the application of the rules presented in Chapter 1 to more advanced examples, such as parameterized programs and multiple resource-allocation paradigms. An example of a parameterized program that can be verified by the techniques presented here is a network of  $N$  processes connected in a ring. This system can be verified uniformly, presenting a single proof that is valid for any value of  $N > 1$ . We also present an algorithm for the automatic construction of linear invariants. These are invariants stating that a linear expression over the program's data and control variables has a constant value throughout the computation. In many cases, these linear invariants are sufficient to establish interesting safety properties of a given program. We then state and prove the (relative) completeness of the verification rules presented in Chapter 1. This implies that if  $p$  is an invariant of a program, then its invariance can always be formally proven using the presented verification rules. Chapter 2 ends with an efficient algorithm for checking whether a state invariance is valid over a finite-state system.

Chapter 3 considers *precedence* properties, i.e., properties which state that one event always precedes another. For example, it may state that the event of the gate being lowered (in the railroad crossing example) must always be preceded by the event of a train approaching the intersection. This requirement prevents spurious lowering of the gate. Other examples of precedence properties are absence of unsolicited response (no output without input), order preservation of messages in a buffer, and bounded overtaking. The property of bounded overtaking provides a bound on how many times one process can overtake another process in the

race for obtaining a common resource. The chapter presents a set of verification rules for establishing the validity of such properties over a given program. As in Chapter 2, the last two sections present a proof of (relative) completeness of the verification rules and an algorithm for automatically checking the validity of precedence properties over a finite-state system.

Chapter 4 concludes the treatment of safety properties by presenting rules for verifying properties expressible by the general safety formula  $\square p$ , where  $p$  is a past formula, i.e., a formula that does not refer to the future. We demonstrate the use of past formulas for compositional verification and for the specification and verification of order-preservation properties. The chapter ends by establishing completeness for the presented verification rules, relative to first-order reasoning, followed by a finite-state verification algorithm.

The final chapter of this volume, Chapter 5, presents a universal finite-state verification algorithm. Although individual algorithms are presented at the end of each chapter for proving that chapter's properties, these algorithms only apply to formulas that have the canonical form for the properties discussed in the chapter. The universal algorithm of Chapter 5 can be applied to an arbitrary temporal formula.

## Teaching

This volume can be taught at different levels of detail, depending on the amount of time the instructor wishes to devote to temporal logic and its use for specification and verification. Shorter teaching programs can be obtained by omitting some material which is less essential to the understanding of the overall picture.

To aid in the design of shorter teaching tracks, we present in Fig. B a dependence tree, showing the dependence of each section on the preceding ones.

## Recommended Tracks

To illustrate possible teaching tracks, we list several possible tracks corresponding to various levels and topics.

- The *Basic (B)* track — Chapter 0, Sections 1.1–1.4, and 3.1–3.3.
- The *Extended (E)* track — Sections 1.5 and 3.4.
- The *Multi-Process Programs (M)* track — Sections 2.1–2.3.
- The *Completeness (C)* track — Sections 2.5, 3.5, and 4.9.
- The *Algorithmic (A)* track — Sections 2.4, 2.6, 3.6, 4.10, and 5.1–5.5.
- The *Past (P)* track — Sections 4.1–4.8, optionally 4.9 and 4.10.

As can be seen from the dependence tree, all sections beyond Chapters 0 and 1 depend on these chapters, which implies that these chapters must be covered in

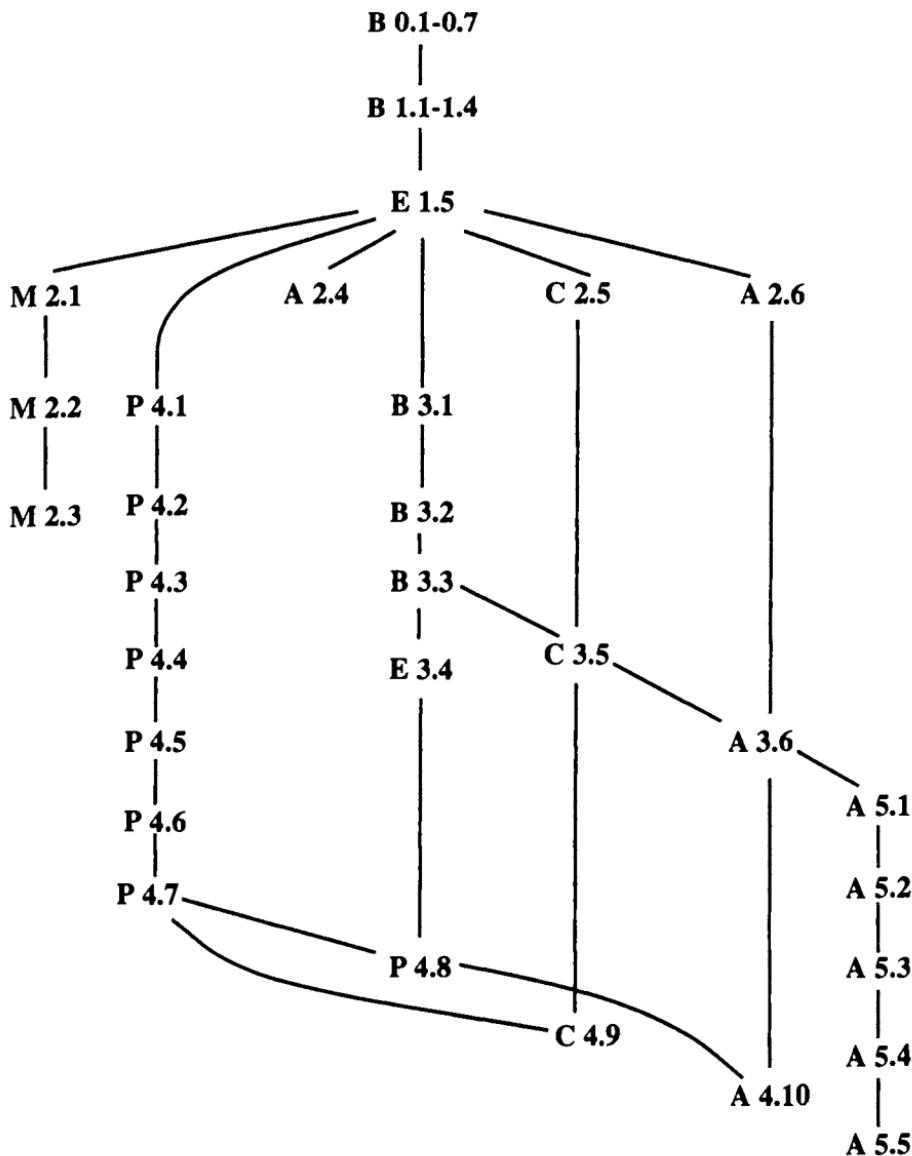


Fig. B. A dependency tree.

any teaching program. Some of the sections (sections 3.5, 3.6, 4.9, and 4.10) are annotated by \* to indicate that they contain a more intricate technical material.

## Problems

Each chapter concludes with a set of problems. Some of the problems are in-

tended to let the readers test their understanding of the material covered in the chapter. Other problems introduce material that was not covered in the chapter. Finally, there are problems that explore alternatives to the way some topics were introduced and developed.

The problems are graded according to their difficulty. Difficult problems are denoted by \*. Research-level problems are denoted by \*\*.

To indicate which problems pertain to a given portion of the text, we annotate the text with references to the appropriate problems, and we provide a page reference with each problem. In solving a problem, readers may use any results that appeared in the text prior to the corresponding page reference. They may also use the results of any previous problem and previous parts of the same problem.

A booklet containing answers to the problems is available to instructors. Please contact the publisher directly.

## Bibliography

Following each chapter, there is a brief bibliographic discussion mentioning some of the research contributions relevant to the topics covered in the chapter. In spite of our sincere effort to refer to all the important works, we may have missed some. We apologize for such omissions and welcome any corrections and comments.

## Support System

The STeP (Stanford Temporal Prover) system, which runs under the UNIX operating system, supports the computer-aided verification of reactive systems based on temporal specifications. The educational version of the system, STeP-E, is based on the techniques presented in this book and can help with the verification exercises. The system is available for those wishing to incorporate STeP-E into their course curriculum.

For information about obtaining the system, please send an e-mail message to “[step-request@cs.stanford.edu](mailto:step-request@cs.stanford.edu).”

## Acknowledgment

We wish to acknowledge the help of many colleagues and students in reading the manuscript in its (almost infinite) number of versions and for their useful comments and suggestions. Particularly helpful suggestions were made by Martin Abadi, Luca de Alfaro, Rajeev Alur, Anuchit Anuchitankul, Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón, Tom Henzinger, Robert Jones, Yonit Kesten, Daphne Koller, Yassine Lakhneche, Narciso Martí-Oliet, Willem-Paul de Roever, Tomás Uribe, Moshe Vardi, and Liz Wolf.

Special thanks are due to Nachum Dershowitz, Avraham Ginzburg, and Henny Sipma, for the thorough reading of earlier drafts of the book and nu-

merous enlightening remarks and constructive criticism. Arjun Kapur has been of special assistance in the preparation of problems for this volume.

We would like to thank the students at Stanford University and the Weizmann Institute who took courses based on this book, for their detailed comments and helpful criticisms.

For support of the research behind this book, we thank the Air Force Office of Scientific Research, the Department of Defense Advanced Research Projects Agency, the National Science Foundation, and the European Community ESPRIT project.

Sarah Fliegelmann has done a magnificent job of typesetting the book.

Eddie Chang and Eric Muller spent long hours patiently helping with the preparation of the computer-generated diagrams.

We are particularly grateful to Carron Kirkwood for the cover design.

Stanford University  
Weizmann Institute

Z.M.  
A.P.



# **Contents**

<b>Preface</b>	vii
<b>Chapter 0: Preliminary Concepts</b>	1
0.1 Fair Transition System	2
0.2 A Programming Language (SPL): Syntax	5
0.3 A Programming Language (SPL): Semantics	18
0.4 Modules	36
0.5 Temporal Logic	42
0.6 Specification of Properties	54
0.7 Overview of the Verification Framework	63
Problems	66
Bibliographic Remarks	74
<b>Chapter 1: Invariance: Proof Methods</b>	81
1.1 Preliminary Notions	81
1.2 Invariance Rule	87
1.3 Finding Inductive Assertions: The Bottom-Up Approach	104
1.4 Finding Inductive Assertions: The Top-Down Approach	111
1.5 Refining Invariants	127
Problems	152
Bibliographic Remarks	164
<b>Chapter 2: Invariance: Applications</b>	167
2.1 Parameterized Programs	168
2.2 Single-Resource Allocation	179
2.3 Multiple-Resource Allocation	191
2.4 Constructing Linear Invariants	201
2.5 Completeness	220

2.6 Finite-State Algorithmic Verification Problems	227
Bibliographic Remarks	232
<b>Chapter 3: Precedence</b>	251
3.1 Waiting-for Rule	251
3.2 Nested Waiting-for Rule	264
3.3 Verification Diagrams	272
3.4 Overtaking Analysis for a Resource Allocator	280
* 3.5 Completeness	288
* 3.6 Finite-State Algorithmic Verification Problems	297
Bibliographic Remarks	307
	314
<b>Chapter 4: General Safety</b>	317
4.1 Invariance Rule for Past Formulas	318
4.2 Applications of the Past Invariance Rule	327
4.3 Compositional Verification	336
4.4 Causality Rule	342
4.5 Backward Analysis	347
4.6 Order-Preservation Properties	354
4.7 History Variables	357
4.8 Back-to Rule	363
* 4.9 Completeness	372
* 4.10 Finite-State Algorithmic Verification Problems	381
Bibliographic Remarks	392
	396
<b>Chapter 5: Algorithmic Verification of General Formulas</b>	399
5.1 Satisfiability of a Temporal Formula	400
5.2 Satisfiability over a Finite-State Program	422
5.3 Validity over a Finite-State Program: Examples	434
5.4 Incremental Tableau Construction	443
5.5 Particle Tableaux	451
Problems	460
Bibliographic Remarks	462
<b>References</b>	465
<b>Index to Symbols</b>	481
<b>General Index</b>	489

## *Chapter 0*

# *Preliminary Concepts*

This book presents methods for verifying that reactive programs satisfy their specifications, where the specifications are formulated in temporal logic.

Reactive programs are programs whose role is to maintain an ongoing interaction with their environment, rather than produce a final value on termination. Examples of reactive programs are operating systems and programs controlling mechanical devices such as an airplane or a train, or ongoing processes such as a nuclear reactor. Some reactive programs are not expected to terminate. Reactive programs must be specified in terms of their ongoing behavior, and temporal logic provides an expressive and natural language for specifying this behavior.

Reactive programs are closely related to concurrency. Many reactive programs consist of parallel processes that run concurrently. Even when the program itself is sequential, it is often helpful to view the complete system as consisting of two processes, the program being one of them and the environment with which it interacts being the other.

In this chapter, we review some of the basic notions introduced in the SPEC book that are used in this volume. A few definitions have been slightly altered to simplify their use for verification. The purpose of this chapter is to make this volume self-contained to a large extent.

Section 0.1 introduces fair transition systems, which are the computational model used to represent reactive programs and systems. In Section 0.2, we present the syntax of a simple programming language that allows concurrency and communication. The semantics of the programming language is defined in Section 0.3 by mapping programs into fair transition systems. While Sections 0.2 and 0.3 consider entire programs, Section 0.4 extends the syntax and semantics to *modules*, which are components of an entire program.

In Section 0.5 we present temporal logic, which is the language used for specifying properties of reactive programs. Section 0.6 presents a hierarchy of

properties, classified according to the formulas expressing them.

## 0.1 Fair Transition System

We assume that all variables that describe states of programs or that appear in formulas specifying properties of programs are taken from a universal set of variables  $\mathcal{V}$ , called the *vocabulary*. Variables in  $\mathcal{V}$  are typed, where the type of a variable, such as *boolean*, *integer*, etc., indicates the domain over which the variable ranges. In particular, we assume that, for each  $x \in \mathcal{V}$ , its *primed version*  $x'$  is also in  $\mathcal{V}$ .

To express the syntax of a fair transition system, we assume an underlying first-order language over  $\mathcal{V}$ , which consists of the following elements:

- *Expressions* are constructed from the variables of  $\mathcal{V}$  and constants (such as 0) to which functions (such as +) are applied.
- *Atomic Formulas* consist of propositions (boolean variables) and formulas constructed by applying predicates (such as  $>$ ) to expressions.
- *Assertions (state formulas)* are constructed by applying boolean connectives and quantification to atomic formulas, constructing first-order formulas.

We define a *state*  $s$  to be a type-consistent interpretation of  $\mathcal{V}$ , assigning to each variable  $u \in \mathcal{V}$  a value  $s[u]$  over its domain. We denote by  $\Sigma$  the set of all states.

### The System

A *fair transition system*  $\langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ , intended to represent a reactive program, is given by the following components:

- $V = \{u_1, \dots, u_n\} \subseteq \mathcal{V}$  : A finite set of *system variables*. Some of these variables represent *data variables*, which are explicitly manipulated by the program text. Other variables are *control variables*, which represent, for example, the location of control in each of the processes in a concurrent program.
- $\Theta$  : The *initial condition*. This is a satisfiable assertion characterizing all the initial states, i.e., states at which the computation of the program can start. A state is defined to be *initial* if it satisfies  $\Theta$ .
- $\mathcal{T}$  : A finite set of *transitions*. Each transition  $\tau \in \mathcal{T}$  is a function

$$\tau: \Sigma \mapsto 2^\Sigma,$$

mapping each state  $s \in \Sigma$  into a (possibly empty) set of states  $\tau(s) \subseteq \Sigma$ . Each state in  $\tau(s)$  is called a  $\tau$ -*successor* of  $s$ .

We say that the transition  $\tau$  is *enabled* on the state  $s$  if  $\rho_\tau(V, V') \neq \emptyset$ . Otherwise, we say that  $\tau$  is *disabled* on  $s$ .

- $\mathcal{J} \subseteq \mathcal{T}$ : A set of *just* transitions (also called *weakly fair* transitions). Informally, the requirement of justice for  $\tau \in \mathcal{J}$  disallows a computation in which  $\tau$  is continually enabled but not taken beyond a certain point.
- $\mathcal{C} \subseteq \mathcal{T}$ : A set of *compassionate* transitions (also called *strongly fair* transitions). Informally, the requirement of compassion for  $\tau \in \mathcal{C}$  disallows a computation in which  $\tau$  is enabled infinitely many times but taken only finitely many times.

Each transition  $\tau \in \mathcal{T}$  is represented by a first-order formula  $\rho_\tau(V, V')$ , called the *transition relation*, which may refer to both unprimed and primed versions of the system variables. The purpose of the transition relation  $\rho_\tau$  is to express the relation holding between a state  $s$  and any of its  $\tau$ -successors  $s' \in \tau(s)$ . We use the unprimed version to refer to values in  $s$ , and the primed version to refer to values in  $s'$ . For example, the formula  $x' = x + 1$  states that  $s'[x]$ , the value of  $x$  in  $s'$ , is greater by 1 than  $s[x]$ , the value of  $x$  in  $s$ .

Thus, the state  $s'$  is a  $\tau$ -successor of the state  $s$  if the formula  $\rho_\tau(V, V')$  evaluates to  $\top$  (true), when we interpret each  $x \in V$  as  $s[x]$ , and its primed version  $x'$  as  $s'[x]$ .

Using the transition relation  $\rho_\tau$ , we can express the enabledness of the transition  $\tau$  by the formula

$$En(\tau): \exists V: \rho_\tau(V, V'),$$

which is true at  $s$  iff  $s$  has some  $\tau$ -successor.

We require that one of the transitions in  $\mathcal{T}$  be the *idling* transition  $\tau_I$  (also called the *stuttering* transition), whose transition relation is  $\rho_I: (V = V')$ . Thus,  $s'$  is a  $\tau_I$ -successor of  $s$  iff  $s$  agrees with  $s'$  on all system variables. We refer to the non-idling transitions as the *diligent* transitions. We denote by  $\mathcal{T}^-$  the set of all diligent transitions.

## Computations

Assume a system  $P$  for which the above components have been specified. Consider

$$\sigma: s_0, s_1, s_2, \dots,$$

an infinite sequence of states. We say that transition  $\tau \in \mathcal{T}$  is *enabled at position  $k$*  of  $\sigma$  if  $\tau$  is enabled on  $s_k$ . We say that the transition  $\tau$  is *taken at position  $k$*  if  $s_{k+1}$  is a  $\tau$ -successor of  $s_k$ . Note that several different transitions can be considered as taken at the same position. This is possible when both  $\rho_{\tau_1}$  and  $\rho_{\tau_2}$  are satisfied by  $(s_k, s_{k+1})$ , for  $\tau_1 \neq \tau_2$ .

The sequence  $\sigma$  is defined to be a *computation* of  $P$  ( *$P$ -computation*) if it satisfies the following requirements:

- *Initiality:*  $s_0$  is initial.
- *Consecution:* For each  $j = 0, 1, \dots$ , the state  $s_{j+1}$  is a  $\tau$ -successor of the state  $s_j$ , i.e.,  $s_{j+1} \in \tau(s_j)$  for some  $\tau \in \mathcal{T}$ .
- *Justice:* For each transition  $\tau \in \mathcal{J}$ , it is not the case that  $\tau$  is continually enabled beyond some position  $j$  in  $\sigma$  (i.e.,  $\tau$  is enabled at every position  $k \geq j$ ) but not taken beyond  $j$ .
- *Compassion:* For each transition  $\tau \in \mathcal{C}$ , it is not the case that  $\tau$  is enabled at infinitely many positions in  $\sigma$  but taken at only finitely many positions.

We refer to the requirements of justice and compassion as *fairness requirements*.

We say that a state  $s$  is  *$P$ -accessible* if it appears in some computation of  $P$ . Clearly, any  $\tau$ -successor of a  $P$ -accessible state is also  $P$ -accessible.

A sequence  $\sigma$  that satisfies the requirements of initiality and consecution, but not necessarily the requirements of justice and compassion, is called a *run* of  $P$ . Obviously, every computation is a run but not vice-versa. Every finite prefix of a run is also a prefix of some computation; thus, every state  $s$  that appears in a run of  $P$  is  $P$ -accessible.

Note that the definitions of  $\tau$ -successor and of computation only restrict the interpretation given to the system variables  $V$ . The interpretation of variables in  $V - V$  is completely unrestricted.

## Variants of States and Computations

Let  $U \subseteq V$  be a set of variables. We say that state  $\hat{s}$  is a  *$U$ -variant of state  $s$*  if  $\hat{s}$  and  $s$  agree on the interpretation of all variables in  $V - U$ , that is,  $\hat{s}[x] = s[x]$  for every  $x \in V - U$ . A state sequence  $\hat{\sigma}: \hat{s}_0, \hat{s}_1, \dots$  is called a  *$U$ -variant of the state sequence  $\sigma: s_0, s_1, \dots$*  if  $\hat{s}_i$  is a  $U$ -variant of  $s_i$  for each  $i = 0, 1, \dots$ .

In the following, let  $U$  be a set of variables that is disjoint from  $V$ , the system variables of system  $P$ . That is,  $U \cap V = \emptyset$ . Since the notion of state  $s_{i+1}$  being the  $\tau$ -successor of  $s_i$  only depends on the interpretation  $s_i$  and  $s_{i+1}$  give to  $V$ , the following statements follow from the definitions:

- If state  $s_{i+1}$  is a  $\tau$ -successor of state  $s_i$  then every  $\hat{s}_{i+1}$ , a  $U$ -variant of  $s_{i+1}$ , is a  $\tau$ -successor of every  $\hat{s}_i$ , a  $U$ -variant of  $s_i$ .
- If  $\sigma: s_0, s_1, \dots$  is a  $P$ -computation then every  $\hat{\sigma}: \hat{s}_0, \hat{s}_1, \dots$ , a  $U$ -variant of  $\sigma$ , is also a  $P$ -computation.
- If  $s$  is a  $P$ -accessible state then so is every  $\hat{s}$ , a  $U$ -variant of  $s$ .

When  $U$  consists of a single variable  $u$ , we refer to  $U$ -variants of states and

sequences simply as  $u$ -variants.

## 0.2 A Programming Language (SPL): Syntax

We introduce a simple concurrent programming language in which example programs will be written. We sometimes refer to this language as **SPL** (Simple Programming Language). The design of the syntax and semantics of SPL reflects a compromise between maximal expressivity and simplicity of presentation.

A program in the language is constructed out of statements, which may be labeled.

### Basic Statements

These statements represent the most basic steps in the computation.

- *Skip*: A trivial do-nothing statement is  
 $\text{skip}$
- *Assignment*: For a list of variables  $u_1, \dots, u_k$  and a list of expressions  $e_1, \dots, e_k$  of corresponding types,  
 $(u_1, \dots, u_k) := (e_1, \dots, e_k)$

is an *assignment* statement. We sometimes denote the multiple assignment by  $\bar{u} := \bar{e}$ . When  $k = 1$ , we write simply  $u := e$ .

- *Await*: For a boolean expression  $c$ ,  
 $\text{await } c$

is an *await* statement. We refer to condition  $c$  as the *guard* of the statement. Execution of “*await*  $c$ ” changes no variables. Its sole purpose is to wait until  $c$  becomes true, at which point the statement terminates, allowing the execution of subsequent statements.

We introduce the statement *halt* as an abbreviation for *await F* (*F* stands for *false*). Obviously, a process reaching this statement cannot progress beyond it.

The following two statements, called *communication statements*, support communication by message passing. These statements refer to a *channel* which is a system variable of a special kind. As with other variables, each channel is associated with a type, identifying the type of values that can be sent via the channel.

- *Send*: For a channel  $\alpha$  and expression  $e$  of compatible type,

$$\alpha \Leftarrow e$$

is a *send* statement. The expression  $e$  is evaluated and its value sent via channel  $\alpha$ .

- *Receive:* For a channel  $\alpha$  and variable  $u$  of compatible type,

$\alpha \Rightarrow u$

is a *receive* statement. This statement causes a value (message) to be read from channel  $\alpha$  and placed in variable  $u$ . If no value is currently available on the channel, execution of this statement is delayed.

The following two statements support semaphore operations.

- *Request:* For an integer variable  $r$ ,

**request  $r$**

is a *request* statement. This statement can be executed only when  $r$  has a positive value. When executed, it decrements  $r$  by 1.

- *Release:* For an integer variable  $r$ ,

**release  $r$**

is a *release* statement. Execution of this statement increments  $r$  by 1.

## Schematic Statements

The following statements provide schematic representations of segments of code that appear in programs for solving the mutual-exclusion or producer-consumer problems. Typically, we are not interested in the internal details of this code but only in its overall behavior concerning termination.

- *Noncritical:*

**noncritical**

is a *noncritical* statement. This statement represents the noncritical activity in programs for mutual exclusion. It is not required that this statement terminate.

The name “noncritical” given to this statement is appropriate for programs that deal with critical sections. We introduce *idle* as synonymous with “noncritical” and will use this name in programs where the notion of critical sections is irrelevant.

- *Critical:*

**critical**

is a *critical* statement. This statement represents the critical activity in programs for mutual exclusion, where coordination between the processes is required. It is required that this statement terminate.

- *Produce:* For an integer variable  $x$ ,

**produce**  $x$

is a *produce* statement. This terminating statement represents the production activity in producer-consumer programs. Its effect is to assign a nonzero value to variable  $x$ , representing the produced value.

- *Consume*: For an integer variable  $y$ ,

**consume**  $y$

is a *consume* statement. This terminating statement represents the consumption activity in producer-consumer programs.

It is assumed that none of the schematic statements modify any of the program variables, except for the production variable  $x$ , explicitly mentioned in the corresponding statements.

## Compound Statements

Compound statements consist of a controlling frame applied to one or more sub-statements, to which we refer as the *children* of the compound statement.

- *Conditional*: For statements  $S_1$  and  $S_2$  and a boolean expression  $c$ ,

**if**  $c$  **then**  $S_1$  **else**  $S_2$

is a *conditional* statement. Its intended meaning is that the boolean condition  $c$  is evaluated and tested. If the condition evaluates to T (true), statement  $S_1$  is selected for subsequent execution; otherwise, if the condition evaluates to F (false),  $S_2$  is selected. Thus, the first step in an execution of the conditional statement is the evaluation of  $c$  and the selection of  $S_1$  or  $S_2$  for further execution. Subsequent steps continue to execute the selected substatement.

When control reaches a conditional statement, the first step in its execution can always be taken. This is because  $c$  always evaluates to either T or F and the first step in an execution of the statement is therefore defined for both values of the condition  $c$ . In contrast, the first step in an execution of **await**  $c$  can be taken only if  $c$  evaluates to T.

A special case of the conditional statement is the *one-branch-conditional* statement

**if**  $c$  **then**  $S_1$ .

Execution of this statement in the case that  $c$  evaluates to F terminates in one step.

- *Concatenation*: For statements  $S_1, \dots, S_k$ ,

$S_1; \dots; S_k$

is a *concatenation* statement. Its intended meaning is sequential execution of

statements  $S_1, \dots, S_k$  one after the other. Thus, the first step in an execution of the concatenation statement is the first step in an execution of  $S_1$ . Subsequent steps continue to execute the rest of  $S_1$ , and when  $S_1$  terminates, proceed to execute  $S_2, \dots, S_k$ . In a program presented as a multi-line text, we often omit the separator ';' at the end of a line.

With concatenation, we can define the *when* statement

**when**  $c$  **do**  $S$

as a synonym for the concatenation

**await**  $c;$   $S$ .

- *Selection:* For statements  $S_1, \dots, S_k$ ,

$S_1$  **or**  $\dots$  **or**  $S_k$

is a *selection* statement. Its intended meaning is a nondeterministic selection of a statement  $S_i$  and its execution. The first step in the execution of the selection statement selects a statement  $S_i$ ,  $i = 1, \dots, k$ , that is currently enabled (ready to be executed) and performs the first step in the execution of  $S_i$ . Subsequent steps proceed to execute the rest of the selected substatement, ignoring the other  $S_j$ 's. If more than one of  $S_1, \dots, S_k$  is enabled, the selection is nondeterministic. If none of the branches are enabled, execution of the selection statement is delayed.

The *selection* statement is often applied to *when* statements. This combination leads to *conditional selection*. For example, the general conditional statement of the *guarded command* language of the form

**if**  $c_1 \rightarrow S_1 \quad \square \quad c_2 \rightarrow S_2 \quad \square \quad \dots \quad \square \quad c_k \rightarrow S_k \quad \text{fi},$

which uses  $\square$  as a separator between alternatives, can be represented in our language by a *selection* statement formed out of *when* statements:

[**when**  $c_1$  **do**  $S_1$ ] **or** [**when**  $c_2$  **do**  $S_2$ ] **or**  $\dots$  **or** [**when**  $c_k$  **do**  $S_k$ ]].

- *While:* For a boolean expression  $c$  and a statement  $S$ ,

**while**  $c$  **do**  $S$

is a *while* statement. Its execution begins by evaluating  $c$ . If  $c$  evaluates to F, execution of the statement terminates. Otherwise, subsequent steps proceed to execute  $S$ . When  $S$  terminates, execution of the *while* statement repeats.

We introduce the notation

**loop forever do**  $S$

as a synonym for

**while** T **do**  $S$ .

Another useful abbreviation is the *for* statement

**for**  $i := 1$  **to**  $m$  **do**  $S$ ,

which is an abbreviation for the concatenation

$i := 1; \text{ while } i \leq m \text{ do } [S; i := i + 1].$

- **Cooperation:** For statements  $S_1, \dots, S_k$ ,

$\ell: [\ell_1: S_1; \hat{\ell}_1: ] \parallel \dots \parallel [\ell_k: S_k; \hat{\ell}_k: ]; \hat{\ell}:$

is a *cooperation* statement. Its intended meaning is the parallel execution of  $S_1, \dots, S_k$ . The first step in the execution of a cooperation statement is referred to as the *entry* step. It can be viewed as setting the stage for the parallel execution of  $S_1, \dots, S_k$  while moving from  $\ell$  to  $\ell_1, \dots, \ell_k$ . Subsequent steps proceed to perform steps from  $S_1, \dots, S_k$ . When all  $S_1, \dots, S_k$  have terminated, there is an additional *exit* step that closes the parallel execution, and moves from  $\hat{\ell}_1, \dots, \hat{\ell}_k$  to  $\hat{\ell}$ . Each parallel component is assigned an *entry label*  $\ell_i$  and an *exit label*  $\hat{\ell}_i$ , which is the location of control after execution of  $S_i$  terminates. Label  $\hat{\ell}_i$  can be viewed as labeling an empty statement following  $S_i$ . We refer to component  $[\ell_i: S_i; \hat{\ell}_i: ]$  as a *process* of the cooperation statement, and say that statements  $S_1, \dots, S_k$  are parallel to one another. A more precise definition of labels and their semantic role will be presented later.

It is important to note that in the combination

$([\ell_1: S_1; \hat{\ell}_1: ] \parallel [\ell_2: S_2; \hat{\ell}_2: ]); S_3,$

execution of  $S_3$  cannot start until both  $S_1$  and  $S_2$  have terminated.

Execution of the cooperation statement, once it has been initialized, proceeds by *interleaving*. That is, steps from the various processes  $S_1, \dots, S_k$  are chosen and executed, one at a time. The justice requirement ensures that a computation does not consistently ignore one of the processes forever.

- **Block:** For a statement  $S$ ,

$[\text{local declaration}; S]$

is a *block* statement. Statement  $S$  is called the *body* of the block.

A *local declaration* is a list of *declaration statements* of the form

**local** variable, ..., variable: type **where**  $\varphi_i$ .

The local declaration identifies the variables that are local to the block, specifies their type, and optionally specifies their initial values.

The *type* of a variable identifies the domain over which the variable ranges and the operations that may be applied to it. We use *basic types* such as **boolean**, **integer**, and **character**, as well as *structured types* such as **array**, **list**, and **set**.

The assertion  $\varphi_i$  appearing in the *where* clause of a declaration statement is of the

form  $y_1 = e_1, \dots, y_n = e_n$ , where  $y_1, \dots, y_n$  are some of the variables declared in this statement and  $e_1, \dots, e_n$  are expressions that may depend only on variables declared outside of the block. The *where* clause may be omitted when no initial values are specified.

The intended meaning of the *where* part is that  $e_1, \dots, e_n$  are the initial values of the variables  $y_1, \dots, y_n$  at the beginning of the computation of the program. Thus, the initialization associated with the *where* part of variables that are declared local to a block is *static*. This means that these variables are initialized only once, in the beginning of the computation. This primitive notion of static initialization is adopted to simplify the representation of programs as fair transition systems.

To ensure unambiguous parsing of statements, we require that every compound statement that appears as a substatement of a larger statement be enclosed within brackets. Thus, the statement  $[S_1; S_2]; S_3$  is a concatenation of statement  $S_1; S_2$  with statement  $S_3$ , while  $S_1; S_2; S_3$  is a concatenation of the three statements  $S_1$ ,  $S_2$ , and  $S_3$ . The two concatenations have the same computational behavior, but syntactically, they are considered different statements.

When we write statements (and programs) in a multi-line format, we relax the above requirement and omit brackets from substatements occupying a full line. We also omit semicolons following such substatements. For example, in Fig. 0.1 (presented in page 13), we omitted the brackets from the two branches of selection statement  $\ell_2$ , even though each of them is a concatenation. We also omitted the semicolon that should have separated the while statement  $\ell_1$  from the assignment statement  $\ell_7$ .

## **Composite and Grouped Statements**

In some cases it is necessary to execute a compound statement  $S$  in one atomic step (taking a single transition) with no interference from parallel statements. To denote that  $S$  is to be executed in such atomic manner, we write  $\langle S \rangle$  and refer to this statement as a *grouped statement*.

To simplify the discussion, we restrict the grouping operation to simple compound statements. Therefore, we first define the notion of a *composite statement* as follows:

- Skip, assignment, await, send, and receive statements are (basic) composite statements.
- If  $S, S_1, \dots, S_k$  are composite statements, then so are  
 $\text{when } c \text{ do } S, \text{ if } c \text{ then } S_1 \text{ else } S_2, [S_1 \text{ or } \dots \text{ or } S_k],$  and  $[S_1; \dots; S_k]$ .

If  $S$  is a composite statement, then

$\langle S \rangle$

is a *grouped statement*.

We further restrict the occurrences of communication statements within grouped statements. A grouped statement that contains a synchronous communication statement (synchronous send or receive) must be of the form

$$\langle S_1; C; S_2 \rangle,$$

where  $C$  is the synchronous communication statement, and  $S_1, S_2$  are statements that do not contain any communication substatements. Statements  $S_1$  and  $S_2$  are optional, and any of them may be omitted.

For asynchronous communication statements, the requirements are less stringent and we only require that no two asynchronous communication substatements address the same channel. Thus,  $\langle \alpha \Leftarrow 1; \alpha \Leftarrow 2 \rangle$  is an inadmissible grouped statement, while  $\langle \alpha \Leftarrow 1; \beta \Leftarrow 2 \rangle$  is admissible.

## Programs

A program  $P$  consists of a declaration followed by a cooperation statement, in which processes may be named.

$$P ::= [\text{declaration}; [P_1 :: [\ell_1: S_1; \hat{\ell}_1: ] \parallel \dots \parallel P_k :: [\ell_k: S_k; \hat{\ell}_k: ]]]$$

The names of the program and of the processes are optional, and may be omitted. We refer to the cooperation statement as the *body* of the program, and to  $P_1, \dots, P_k$  as the *top-level* processes of  $P$ . In contrast, processes that are children of cooperation statements contained in  $S_1, \dots, S_k$  are called *internal processes*.

A *declaration* consists of a sequence of *declaration statements* of the form

$$\text{mode variable, } \dots, \text{variable: type where } \varphi_i.$$

Each declaration statement identifies the mode and type of a list of variables and, optionally, specifies constraints on their initial values.

The *mode* of each declaration statement may be one of the following:

**in** — Specifies variables that are inputs to the program.

**local** — Specifies variables that are local to the program. These variables are used in the execution of the program, but are not recognized outside the program.

**out** — Specifies variables that are outputs of the program.

A statement  $S$  may refer to a variable only if the variable is declared at the head of the program or at the head of a block containing  $S$ .

The program is not allowed to modify (i.e., assign new values to) variables that are declared as in variables. The distinction between modes **local** and **out**

is mainly intended to help in understanding the program and has no particular formal significance.

The optional assertion  $\varphi_i$  imposes constraints on the initial values of the variables declared in this statement. For *local* and *out* variables it is restricted to the form  $y = e$ , specifying an initial value. The expression  $e$  may only depend on *in* variables. For *in* variables, assertion  $\varphi_i$  may be any constraint on the range of values expected for these variables.

Let  $\varphi_1, \dots, \varphi_n$  be the assertions appearing in the declaration statements of a program. We refer to the conjunction  $\varphi: \varphi_1 \wedge \dots \wedge \varphi_n$  as the *data-precondition* of the program. We refer to the variables declared in the program as the *program variables*.

Channels are declared by a *channel declaration* that may assume one of the two forms:

mode  $\alpha_1, \alpha_2, \dots, \alpha_n$ : channel of type

mode  $\alpha_1, \alpha_2, \dots, \alpha_n$ : channel [1..] of type where  $\varphi_i$ .

These declarations identify  $\alpha_1, \dots, \alpha_n$  as channels through which messages of the specified type can be sent and received. Channels defined by a declaration of the first form are called *synchronous channels*. Such channels have no buffering capacity and a communication through them can occur only when both sender and receiver execute their statements at the same step.

Channels defined by a declaration of the second form are called *asynchronous channels*. These channels have unbounded buffering capacity. This means that a sender can send an unbounded number of messages before the receiver reads the first one. The declaration of asynchronous channels may specify an initial condition on the value of the channel's buffer. If no initial condition is explicitly specified, we assume the standard initial condition  $\alpha_i = \Lambda$  for each declared asynchronous channel. This initial condition specifies that the channel's buffer is initially empty.

## Labels and Locations

We assume that each statement in the program has a label. In the presentation of programs, we often omit some of the labels and retain only those to which we refer in discussions and proofs. To illustrate this point, consider the fully labeled program GCD-F (for computing the greatest common divisor) of Fig. 0.1 whose entry and exit labels are  $\ell_0$  and  $\ell_8$ , respectively.

Labels fulfill two roles in the analysis of programs. The first role is to provide unique identification for statements. The second role is to serve as locations of control. We envisage control to reside at the label before proceeding to perform the next execution step.

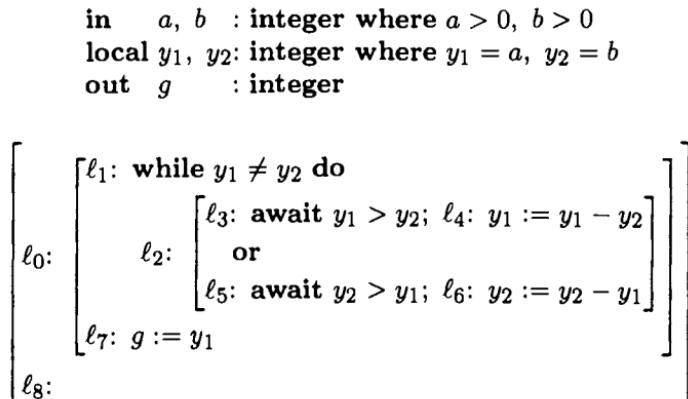


Fig. 0.1. A fully labeled program GCD-F.

Examining the program of Fig. 0.1, we see that  $\ell_2$ ,  $\ell_3$ , and  $\ell_5$  label distinct statements, but, on the other hand, whenever control is at  $\ell_2$  it is also at  $\ell_3$  and at  $\ell_5$  at the same time. This is because while at  $\ell_2$ , the program contemplates a selection between  $\ell_3$  and  $\ell_5$ , which are both candidates for execution. The actual selection between statements  $\ell_3$  and  $\ell_5$  is done only when control moves to  $\ell_4$  or to  $\ell_6$ . This shows that distinct labels may point to the same control location.

### The Equivalence Relation $\sim_L$

To overcome this redundancy, we introduce an *equivalence relation* between labels, denoted  $\sim_L$ . This relation is determined by the following rules:

- For a concatenation statement  $\ell: [\ell_1: S_1; \dots; \ell_k: S_k]$ , we have  

$$\ell \sim_L \ell_1.$$
- For a selection statement  $\ell: [\ell_1: S_1 \text{ or } \dots \text{ or } \ell_k: S_k]$ , we have  

$$\ell \sim_L \ell_1 \sim_L \dots \sim_L \ell_k.$$
- For a block  $\ell: [\text{local declaration}; \ell_1: S_1]$ , we have  

$$\ell \sim_L \ell_1.$$

No equivalence is associated with the cooperation statement

$$\ell: [[\ell_1: S_1; \widehat{\ell}_1: ] \parallel [\ell_2: S_2; \widehat{\ell}_2: ]].$$

Thus, label  $\ell$  is not considered to be equivalent with either  $\ell_1$  or  $\ell_2$ .

Consider, for example, the fully labeled program presented in Fig. 0.1. Following the rules for label equivalence, we observe that, due to the rules for con-

catenation and selection,  $\ell_0 \sim_L \ell_1$  and  $\ell_2 \sim_L \ell_3 \sim_L \ell_5$ .

We define a *location* to be an equivalence class of labels with respect to the label equivalence relation  $\sim_L$ . For a label  $\ell$ , we denote by  $[\ell]$  the equivalence class containing all the labels that are  $\sim_L$ -equivalent to  $\ell$ . We call  $[\ell]$  the *location corresponding to the label  $\ell$* .

For example, the program presented in Fig. 0.1 has six locations given by

$$\begin{array}{lll} [\ell_0] = \{\ell_0, \ell_1\} & [\ell_4] = \{\ell_4\} & [\ell_7] = \{\ell_7\} \\ [\ell_2] = \{\ell_2, \ell_3, \ell_5\} & [\ell_6] = \{\ell_6\} & [\ell_8] = \{\ell_8\}. \end{array}$$

To simplify the recognition of equivalent labels, we adopt a syntactical convention by which all equivalent labels share a common subscript and may be distinguished by having different superscripts.

In Fig. 0.2 we present a relabeled version of program GCD-F of Fig. 0.1.

```
in    a, b : integer where a > 0, b > 0
local y1, y2: integer where y1 = a, y2 = b
out   g      : integer
```

$\ell_1: \text{while } y_1 \neq y_2 \text{ do}$ $\ell_2: \left[ \begin{array}{l} \ell_2^a: \text{await } y_1 > y_2; \ell_4: y_1 := y_1 - y_2 \\ \text{or} \\ \ell_2^b: \text{await } y_2 > y_1; \ell_6: y_2 := y_2 - y_1 \end{array} \right]$ $\ell_7: g := y_1$ $\ell_8:$	$\boxed{\phantom{\ell_1: \text{while } y_1 \neq y_2 \text{ do}}}$
---	---

Fig. 0.2. A partially labeled program GCD.

In this presentation, we have omitted label  $\ell_0$  and consider  $\ell_1$  to be the entry label. Labels  $\ell_3$  and  $\ell_5$  of Fig. 0.1 are renamed  $\ell_2^a$  and  $\ell_2^b$  in Fig. 0.2 to make their equivalence to  $\ell_2$  explicit and yet provide unique references to the statements they label. We adopt this labeling convention throughout the book.

When there is no danger of confusion, we will refer to locations by the name of a representative label. Thus, the six locations of the program in Fig. 0.2 are represented by

$$\ell_1, \ell_2, \ell_4, \ell_6, \ell_7, \ell_8,$$

where  $\ell_2$  represents the location  $\{\ell_2, \ell_2^a, \ell_2^b\}$ .

### Post-Location of Statements

Every statement  $S$  in the program, excluding the program's body, has a unique location reached by the control when execution of the statement has just terminated. We refer to this location as the *post-location* of statement  $S$ , denoted by  $\text{post}(S)$ . The post-locations of various statements are determined by the following rules:

- For a cooperation statement  $[\ell_1: S_1; \hat{\ell}_1: ] \parallel \dots \parallel [\ell_k: S_k; \hat{\ell}_k: ]$ , we have  
 $\text{post}(S_i) = [\hat{\ell}_i]$ , for every  $i = 1, \dots, k$ .
- For a concatenation statement  $S = [\ell_1: S_1; \dots; \ell_k: S_k]$ , we have  
 $\text{post}(S_i) = [\ell_{i+1}]$ , for  $i = 1, \dots, k - 1$ , and  $\text{post}(S_k) = \text{post}(S)$ .
- For a selection statement  $S = [\ell_1: S_1 \text{ or } \dots \text{ or } \ell_k: S_k]$ , we have  
 $\text{post}(S_1) = \dots = \text{post}(S_k) = \text{post}(S)$ .
- For a conditional statement  $S = [\text{if } c \text{ then } S_1 \text{ else } S_2]$ , we have  
 $\text{post}(S_1) = \text{post}(S_2) = \text{post}(S)$ .
- For a while statement  $[\ell : \text{while } c \text{ do } S']$ , we have  
 $\text{post}(S') = [\ell]$ .
- For a block  $S = [\text{local declaration}; S']$ , we have  
 $\text{post}(S') = \text{post}(S)$ .

If  $\ell$  is the identifying label of statement  $S$ , we may write  $\text{post}(S)$  as  $\text{post}(\ell)$ .

Consider, for example, the program of Fig. 0.2. The post-locations for its statements are given by

$$\begin{aligned}\text{post}(\ell_1) &= [\ell_7] \\ \text{post}(\ell_2) &= \text{post}(\ell_4) = \text{post}(\ell_6) = [\ell_1] \\ \text{post}(\ell_2^a) &= [\ell_4] \\ \text{post}(\ell_2^b) &= [\ell_6] \\ \text{post}(\ell_7) &= [\ell_8].\end{aligned}$$

Note that even though  $\ell_2$  and  $\ell_2^a$  represent the same location  $[\ell_2] = \{\ell_2, \ell_2^a, \ell_2^b\}$ , they label different statements that have different post-locations.

If  $[\ell_i]$  is a post-location of statement  $S$ , we refer to any label  $\ell \in [\ell_i]$  as a *post-label* of  $S$ .

## Parallel and Conflicting Labels

If  $S'$  is a substatement of  $S$ , we say that  $S$  is an *ancestor* of  $S'$ . In particular, every statement is an ancestor of itself. A statement  $S$  is said to be a *common ancestor* of  $S_1$  and  $S_2$  if it is an ancestor of both  $S_1$  and  $S_2$ . Statement  $S$  is defined to be a *least common ancestor* (LCA) of statements  $S_1$  and  $S_2$  if

- $S$  is a common ancestor of  $S_1$  and  $S_2$ , and
- Any other common ancestor  $\tilde{S}$  of  $S_1$  and  $S_2$  is an ancestor of  $S$ .

It is easy to see that every two statements have a unique LCA.

For example, consider the statement

$$[S_1; [S_2 \parallel S_3]; S_4] \parallel S_5.$$

We observe that

the LCA of  $S_2$  and  $S_3$  is  $[S_2 \parallel S_3]$ ;

the LCA of  $S_2$  and  $[S_2 \parallel S_3]$  is  $[S_2 \parallel S_3]$ ;

the LCA of  $S_2$  and  $S_4$  is  $[S_1; [S_2 \parallel S_3]; S_4]$ ;

the LCA of  $S_2$  and  $S_5$  is  $[S_1; [S_2 \parallel S_3]; S_4] \parallel S_5$ .

Statements  $S$  and  $\tilde{S}$  are defined to be *parallel* if their LCA is a cooperation statement that is different from both  $S$  and  $\tilde{S}$ .

Thus, within the context of statement  $S = [S_1; [S_2 \parallel S_3]; S_4] \parallel S_5$ , substatement  $S_2$  is parallel to  $S_3$  because their LCA is  $S_2 \parallel S_3$ . Statement  $S_2$  is also parallel to  $S_5$  because their LCA is the full cooperation statement  $S$ . However,  $S_2$  is not parallel to  $S_4$ , because their LCA is  $[S_1; [S_2 \parallel S_3]; S_4]$ , which is a concatenation and not a cooperation statement.

Two labels are defined to be *parallel* if they label parallel statements. This definition can be extended to locations since, if  $\ell_1$  is parallel to  $\ell_2$  then any label in  $[\ell_1]$  is parallel to any label in  $[\ell_2]$ .

Two inequivalent labels that are not parallel are defined to be *conflicting*. Similarly, two different locations are called conflicting if they are not parallel.

Consider, for example, the statement

$$[\ell_1: S_1; \ell_2: ([\ell_3: S_3; \hat{\ell}_3: ] \parallel [\ell_4: S_4; \hat{\ell}_4: ]); \ell_5: S_5; \hat{\ell}_5: ] \parallel [\ell_6: S_6; \hat{\ell}_6: ].$$

Label  $\ell_3$  is parallel to each of  $\{\ell_4, \hat{\ell}_4, \ell_6, \hat{\ell}_6\}$  and in conflict with each of  $\{\ell_1, \ell_2, \hat{\ell}_3, \ell_5, \hat{\ell}_5\}$ . Labels  $\ell_6$  and  $\hat{\ell}_6$  are in conflict with each other and are parallel to each of  $\{\ell_1, \ell_2, \ell_3, \hat{\ell}_3, \ell_4, \hat{\ell}_4, \ell_5, \hat{\ell}_5\}$ .

## The Limited Critical Reference Requirement

A reference to a variable on the left-hand side of an assignment or to the variable mentioned in a *receive*, *produce*, *request*, or a *release* statement is called a *writing reference*. All other references to variables are considered *reading references*.

A reference to a variable occurring in a statement  $S$  is called *critical* if it is:

- a writing reference to a variable that has reading or writing references in a statement parallel to  $S$ , or
- a reading reference to a variable that has a writing reference in a statement parallel to  $S$ .

In addition, all references to a channel in a *send* or *receive* statement are considered critical.

$$P_1 :: \left[ \begin{array}{l} \ell_1: \boxed{b} := \boxed{b} \cdot y_1 \\ \ell_2: \boxed{y_1} := y_1 - 1 \\ \ell_3: \end{array} \right] \quad || \quad P_2 :: \left[ \begin{array}{l} m_1: \text{await } \boxed{y_1} + y_2 \leq n \\ m_2: \boxed{b} := \boxed{b} / y_2 \\ m_3: y_2 := y_2 + 1 \\ m_4: \end{array} \right]$$

Fig. 0.3. Critical references.

Consider a program containing the statement presented in Fig. 0.3 and no other statement parallel to it. We have drawn squares around the critical references. Note that all reading and writing references to  $b$  are critical, since  $b$  is written by both processes. The writing reference to  $y_1$  in  $P_1$  is critical, since  $P_2$  reads  $y_1$ . Reading references to  $y_1$  are critical in  $P_2$  but not in  $P_1$ , since  $P_1$  is the only process writing  $y_1$ . Neither reading nor writing references to  $y_2$  are critical, since  $P_2$  is the only process referencing  $y_2$ .

A statement is said to obey the *limited critical reference* (LCR) restriction, or be an *LCR-statement*, if each test appearing in the statement (for *await*, *conditional*, and *while* statements) and each assignment statement appearing in it contains at most one critical reference. Thus, statements  $\ell_2$ ,  $m_1$ , and  $m_3$  in the example of Fig. 0.3 are LCR-statements while statements  $\ell_1$  and  $m_2$  violate the LCR restriction.

A program  $P$  is called an *LCR-program* if each statement in  $P$  is an LCR-statement. As explained in the SPEC book, there are several advantages to obeying the LCR restriction. The main advantage is that their implementation on shared-memory architectures is straightforward and does not require additional

protection mechanisms. Therefore, we make an effort to present most of our examples in an LCR form. However, we also use some simple non-LCR programs as examples.

### 0.3 A Programming Language (*SPL*): Semantics

Each program corresponds to a fair transition system. We will identify each of the components of the fair transition system corresponding to a given program.

We refer to this identification as the *transition semantics* of the program. The interpretation of a program as a fair transition system leads to a description of the computations of the program.

#### System Variables

The system variables  $V$  consist of the program variables  $Y = \{y_1, \dots, y_n\}$ , declared in the program, and a single control variable  $\pi$ ; thus  $V = Y \cup \{\pi\}$ . The program variables  $Y$  range over their respectively declared data domains. The control variable  $\pi$  ranges over sets of locations of the program. The value of  $\pi$  in a state denotes all the locations of the program in which control currently resides.

For example, the system variables for program GCD of Fig. 0.2 are  $V: \{\pi, a, b, y_1, y_2, g\}$ , where  $\pi$  ranges over subsets of  $\{[\ell_1], [\ell_2], [\ell_4], [\ell_6], [\ell_7], [\ell_8]\}$ , while all other variables range over the integers.

For a label  $\ell$ , we define the control predicate  $at\_l$  by

$$at\_l: [\ell] \in \pi.$$

In a similar way, we define  $at'_l$  to be

$$at'_l: [\ell] \in \pi'.$$

For each asynchronous channel  $\alpha$  of a declared type  $t$ ,  $Y$  includes a system variable  $\alpha$  whose type is a list of type  $t$ . This variable holds the list of pending messages, i.e., messages sent on  $\alpha$  but not yet received from it.

#### Initial Condition

Consider a program

$$P :: [declaration; [P_1 :: [\ell_1: S_1; \widehat{\ell}_1: ] || \dots || P_k :: [\ell_k: S_k; \widehat{\ell}_k: ]]]$$

Let  $\varphi$  denote the data-precondition of the program, obtained by taking the con-

junction of all the assertions  $\varphi_i$  that appear in the *where* clauses of the declaration. This conjunction includes the default initial value requirement  $\alpha = \Lambda$  for asynchronous channels. The initial condition  $\Theta$  for program  $P$  is defined as

$$\Theta: \pi = \{[\ell_1], \dots, [\ell_k]\} \wedge \varphi.$$

This implies that the first state in an execution of the program begins with the control set to the entry locations of the top-level processes, after all the initialization of the local and output variables has been performed.

For example, the initial condition of program GCD of Fig. 0.2 is given by

$$\Theta: \pi = \{[\ell_1]\} \wedge a > 0 \wedge b > 0 \wedge y_1 = a \wedge y_2 = b.$$

For a program  $P$ , none of whose declaration statements has a *where* part, i.e.,  $\varphi = \top$ , we simply define

$$\Theta: \pi = \{[\ell_1], \dots, [\ell_k]\}.$$

## Transitions

To ensure that every state has some transition enabled on it, we uniformly include the idling transition  $\tau_I$  in the transition system corresponding to each program. The transition relation for  $\tau_I$  is

$$\rho_I: V' = V.$$

We proceed to define the transition relations for the transitions associated with each of the statements in the language. Let  $L$  and  $\widehat{L}$  be two sets of locations. The abbreviation

$$move(L, \widehat{L}): L \subseteq \pi \wedge \pi' = (\pi - L) \cup \widehat{L}$$

expresses a simultaneous move of control from all the locations in  $L$  to the new locations in  $\widehat{L}$ . When there is no danger of confusion, we will write  $move(\{[\ell_1], \dots, [\ell_a]\}, \{\widehat{\ell}_1, \dots, \widehat{\ell}_b\})$  simply as  $move(\{\ell_1, \dots, \ell_a\}, \{\widehat{\ell}_1, \dots, \widehat{\ell}_b\})$ , using labels to represent locations. For most of the transitions, control moves from a single location represented by label  $\ell$  to another location  $\widehat{\ell}$ . We refer to this simple case, in which  $L = \{[\ell]\}$  and  $\widehat{L} = \{\widehat{\ell}\}$  by the notation

$$move(\ell, \widehat{\ell}): move(\{\ell\}, \{\widehat{\ell}\}).$$

Typically, every transition modifies only a few of the system variables. We therefore introduce, for a subset  $U$  of the system variables,  $U \subseteq V$ , the notation

$$pres(U): \bigwedge_{u \in U} (u' = u).$$

The formula  $pres(U)$  states that all variables in  $U$  preserve their value at the current step. Recall that  $V = \{\pi\} \cup Y$ .

In the following, we represent a statement  $S$  with label  $\ell$  and post-label  $\hat{\ell}$  in the form  $[\ell: S; \hat{\ell}:]$ .

## Basic Statements

- *Skip*

With the statement

$$\ell: \text{skip}; \hat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \text{move}(\ell, \hat{\ell}) \wedge \text{pres}(Y).$$

This transition modifies no program variables and simply moves control from  $\ell$  to  $\hat{\ell}$ .

- *Assignment*

With the statement

$$\ell: \bar{u} := \bar{e}; \hat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \text{move}(\ell, \hat{\ell}) \wedge \bar{u}' = \bar{e} \wedge \text{pres}(Y - \{\bar{u}\}).$$

- *Await*

With the statement

$$\ell: \text{await } c; \hat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \text{move}(\ell, \hat{\ell}) \wedge c \wedge \text{pres}(Y).$$

The transition  $\tau_\ell$  is enabled only when control is at  $\ell$  and condition  $c$  holds. When taken, control moves from  $\ell$  to  $\hat{\ell}$ .

The following two definitions deal with communication statements over asynchronous channels.

- *Asynchronous Send*

With the *send* statement

$$\ell: \alpha \Leftarrow e; \hat{\ell}:$$

where  $\alpha$  is an asynchronous channel, we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \text{move}(\ell, \hat{\ell}) \wedge \alpha' = \alpha \bullet e \wedge \text{pres}(Y - \{\alpha\}).$$

The equality  $\alpha' = \alpha \bullet e$  specifies the new value of  $\alpha$  as being obtained by appending the value of  $e$  to the end of  $\alpha$ .

- *Asynchronous Receive*

With the *receive* statement

$$\ell: \alpha \Rightarrow u; \quad \widehat{\ell}: \quad$$

where  $\alpha$  is an asynchronous channel, we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \quad move(\ell, \widehat{\ell}) \wedge |\alpha| > 0 \wedge \alpha = u' \bullet \alpha' \wedge pres(Y - \{u, \alpha\}).$$

This formula states that transition  $\tau_\ell$  is enabled only if channel  $\alpha$  is currently nonempty and, when executed, its effect is to deposit the first element (head) of  $\alpha$  in  $u$  and to remove this element (retaining the tail) from  $\alpha$ .

Since synchronous channels allow no buffering, communication over such channels is possible only by a joint execution of matching *send* and *receive* statements.

Two parallel statements are considered to be *matching* if they form an  $\alpha \Leftarrow e$ ,  $\alpha \Rightarrow u$  pair for some  $e$  and  $u$  and for the same synchronous channel  $\alpha$ . When two matching statements are jointly ready to execute, their execution is atomic and simultaneous and the effect is equivalent to the assignment  $u := e$ . This leads to the following definition.

- *Synchronous Send-Receive*

With each pair of matching *send* and *receive* statements

$$\ell: \alpha \Leftarrow e; \quad \widehat{\ell}: \quad m: \alpha \Rightarrow u; \quad \widehat{m}: \quad$$

we associate a (joint) transition  $\tau_{(\ell,m)}$ , whose transition relation  $\rho_{(\ell,m)}$  is given by

$$\rho_{(\ell,m)}: \quad move(\{\ell, m\}, \{\widehat{\ell}, \widehat{m}\}) \wedge u' = e \wedge pres(Y - \{u\}).$$

Thus, transition  $\tau_{(\ell,m)}$  is enabled only if both *at\_ℓ* and *at\_m* hold simultaneously. When executed, the transition causes joint progress in the two processes containing the communication statements, simultaneously replacing in  $\pi$  location  $[\ell]$  by  $[\widehat{\ell}]$  and location  $[m]$  by  $[\widehat{m}]$ . It also performs the communication itself, assigning to  $u$  the value of the expression  $e$ .

The final definitions deal with semaphore statements.

- *Request*

With the statement

$$\ell: \text{request } r; \quad \widehat{\ell}: \quad$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \quad move(\ell, \widehat{\ell}) \wedge r > 0 \wedge r' = r - 1 \wedge pres(Y - \{r\}).$$

Thus, this statement is enabled when control is at  $\ell$  and  $r$  is positive. When executed, it decrements  $r$  by 1.

- *Release*

With the statement

$\ell$ : **release**  $r$ ;  $\hat{\ell}$ :

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$\rho_\ell$ :  $move(\ell, \hat{\ell}) \wedge r' = r + 1 \wedge pres(Y - \{r\})$ .

This statement increments  $r$  by 1.

## Schematic Statements

- *Noncritical*

With the statement

$\ell$ : **noncritical**;  $\hat{\ell}$ :

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$\rho_\ell$ :  $move(\ell, \hat{\ell}) \wedge pres(Y)$ .

Thus, the only observable action of this statement is to terminate. The situation that execution of the noncritical section does not terminate is modeled by a computation that does not take transition  $\tau_\ell$ . This is allowed by excluding  $\tau_\ell$  from the justice set. The same transition is associated with statement **idle**, which is synonymous with the *noncritical* statement.

- *Critical*

With the statement

$\ell$ : **critical**;  $\hat{\ell}$ :

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$\rho_\ell$ :  $move(\ell, \hat{\ell}) \wedge pres(Y)$ .

The observable action of the *critical* statement is to terminate.

- *Produce*

With the statement

$\ell$ : **produce**  $x$ ;  $\hat{\ell}$ :

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$\rho_\ell$ :  $move(\ell, \hat{\ell}) \wedge x' \neq 0 \wedge pres(Y - \{x\})$ .

The observable action of the *produce* statement is to assign a nonzero value to variable  $x$ . Note that this transition is nondeterministic, that is, a state  $s$  may have more than one  $\tau_\ell$ -successor. In fact it may have infinitely many successors, one for each possible value of  $x'$ .

- *Consume*

With the statement

$$\ell: \text{consume } y; \quad \widehat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: \text{move}(\ell, \widehat{\ell}) \wedge \text{pres}(Y).$$

The observable action of the *consume* statement is to terminate.

## Compound Statements

- *Conditional*

With the statement

$$\ell: [\text{if } c \text{ then } \ell_1: S_1 \text{ else } \ell_2: S_2]; \quad \widehat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation is given by  $\rho_\ell: \rho_\ell^T \vee \rho_\ell^F$ , where

$$\rho_\ell^T: \text{move}(\ell, \ell_1) \wedge c \wedge \text{pres}(Y)$$

$$\rho_\ell^F: \text{move}(\ell, \ell_2) \wedge \neg c \wedge \text{pres}(Y).$$

Relation  $\rho_\ell^T$  corresponds to the case where  $c$  evaluates to true and execution proceeds to  $\ell_1$ , while  $\rho_\ell^F$  corresponds to the case where  $c$  evaluates to false and execution proceeds to  $\ell_2$ .

For the one-branch conditional

$$\ell: [\text{if } c \text{ then } \ell_1: S_1]; \quad \widehat{\ell}:$$

the two disjuncts of the transition relation are

$$\rho_\ell^T: \text{move}(\ell, \ell_1) \wedge c \wedge \text{pres}(Y)$$

$$\rho_\ell^F: \text{move}(\ell, \widehat{\ell}) \wedge \neg c \wedge \text{pres}(Y).$$

Transition  $\tau_\ell$  can be viewed as consisting of two subtransitions  $\tau_\ell^T$  and  $\tau_\ell^F$ , to which we refer as *modes*, and having the transition relations  $\rho_\ell^T$  and  $\rho_\ell^F$ . The relation between  $\tau_\ell$  and its modes can be summarized as follows<sup>1</sup>:

$\tau_\ell$  is enabled on state  $s$       iff    either  $\tau_\ell^T$  or  $\tau_\ell^F$  is enabled on  $s$

$\tau_\ell$  is taken at position  $j \geq 0$     iff    either  $\tau_\ell^T$  or  $\tau_\ell^F$  is taken at  $j$

$s'$  is a  $\tau_\ell$ -successor of state  $s$     iff     $s'$  is either a  $\tau_\ell^T$ -successor or a  $\tau_\ell^F$ -successor of  $s$ .

<sup>1</sup> Our treatment of conditional transitions here differs from that of the SPEC book. In the SPEC book,  $\tau_\ell^T$  and  $\tau_\ell^F$  are considered as separate transitions. Here they are considered as modes of a single transition.

- *While*

With the statement

$$\ell: [\text{while } c \text{ do } [\tilde{\ell}: \tilde{S}]]]; \quad \hat{\ell}:$$

we associate a transition  $\tau_\ell$  (with two modes), whose transition relation is given by  $\rho_\ell: \rho_\ell^T \vee \rho_\ell^F$ , where

$$\rho_\ell^T: \text{move}(\ell, \tilde{\ell}) \wedge c \wedge \text{pres}(Y)$$

$$\rho_\ell^F: \text{move}(\ell, \hat{\ell}) \wedge \neg c \wedge \text{pres}(Y).$$

Note that in the case of a true  $c$  control moves from  $\ell$  to  $\tilde{\ell}$ , according to  $\rho_\ell^T$ , while in the case of a false  $c$  it moves from  $\ell$  to  $\hat{\ell}$ , according to  $\rho_\ell^F$ .

- *Cooperation*

Excluding the body of the program, we associate with each *cooperation* statement,

$$\ell: [[\ell_1: S_1; \hat{\ell}_1] \parallel \cdots \parallel [\ell_k: S_k; \hat{\ell}_k]]]; \quad \hat{\ell}:$$

an *entry transition*  $\tau_\ell^E$  and an *exit transition*  $\tau_\ell^X$ . The corresponding transition relations are given, respectively, by

$$\rho_\ell^E: \text{move}(\{\ell\}, \{\ell_1, \dots, \ell_k\}) \wedge \text{pres}(Y)$$

$$\rho_\ell^X: \text{move}(\{\hat{\ell}_1, \dots, \hat{\ell}_k\}, \{\hat{\ell}\}) \wedge \text{pres}(Y).$$

The entry transition begins execution of the cooperation statement by placing in  $\pi$  the set of locations that are just in front of the parallel statements  $S_1, \dots, S_k$ . The exit transition can be taken only if all the parallel statements have terminated, which is detectable by observing that  $\pi$  satisfies

$$\text{at\_}\hat{\ell}_1 \wedge \cdots \wedge \text{at\_}\hat{\ell}_k.$$

Note that the absence of entry or exit transitions associated with the body of the program is consistent with the fact that, according to the initial condition, execution starts with  $\pi$  set to the entry locations of the top-level processes.

The discerning reader may have observed that no transitions are directly associated with the concatenation, selection, or block statements. This is because each transition occurring in the execution of one of these statements can be attributed to one of their substatements.

When there is no danger of confusion, we will often refer to the transition  $\tau_\ell$  simply as  $\ell$  and to the modes  $\tau_\ell^T$  and  $\tau_\ell^F$  as  $\ell^T$  and  $\ell^F$ , respectively.

In **Problem 0.1**, we request the reader to consider a different version of the *while* statement and assign it a transition relation.

## Grouped Statements

To define the transition semantics of a grouped statement  $\langle S \rangle$ , we first define the notion of the *data (transformation)* relation  $\delta[S]$  associated with a composite statement  $S$ . Like a transition relation, the data relation  $\delta[S]$  describes the relation between a state  $s$  and the state  $s'$  resulting from executing  $S$  on  $s$ . Formula  $\delta[S]$  uses the data variables  $Y = V - \{\pi\}$  in order to refer to their values in  $s$ , and a primed copy  $Y'$  of these variables to refer to their values in  $s'$ .

There are, however, two differences between the data relation  $\delta[S]$  and the transition relation  $\rho_\ell$  for a composite statement  $\ell$ :  $S$ . The first difference is that  $\delta[S]$  completely ignores the control variable  $\pi$  and only considers the data part of the state. The second difference is that, for a compound statement  $S$ ,  $\rho_\ell$  is only concerned with the first step in the execution of  $S$ , while  $\delta[S]$  captures the state transformation effected by execution of the complete  $S$ .

The data relation  $\delta[S]$  is defined inductively as follows:

- *Skip*

$$\delta[\text{skip}]: \quad \text{pres}(Y)$$

- *Assignment*

$$\delta[\bar{u} := \bar{e}]: \quad \bar{u}' = \bar{e} \wedge \text{pres}(Y - \{\bar{u}\})$$

- *Await*

$$\delta[\text{await } c]: \quad c \wedge \text{pres}(Y)$$

- *Asynchronous Send*

$$\delta[\alpha \leftarrow e]: \quad \alpha' = \alpha \bullet e \wedge \text{pres}(Y - \{\alpha\})$$

- *Asynchronous Receive*

$$\delta[\alpha \Rightarrow u]: \quad |\alpha| > 0 \wedge \alpha = u' \bullet \alpha' \wedge \text{pres}(Y - \{u, \alpha\}).$$

- *When*

$$\delta[\text{when } c \text{ do } S]: \quad c \wedge \delta[S]$$

- *Conditional*

$$\delta[\text{if } c \text{ then } S_1 \text{ else } S_2]: \quad (c \wedge \delta[S_1]) \vee (\neg c \wedge \delta[S_2])$$

$$\delta[\text{if } c \text{ then } S]: \quad (c \wedge \delta[S]) \vee (\neg c \wedge \text{pres}(Y))$$

- *Selection*

$$\delta[S_1 \text{ or } \dots \text{ or } S_k]: \quad \delta[S_1] \vee \dots \vee \delta[S_k]$$

- *Concatenation*

$$\delta[S_1; S_2]: \quad \exists \tilde{Y}: \left( \delta[S_1](Y, \tilde{Y}) \wedge \delta[S_2](\tilde{Y}, Y') \right).$$

Thus, assuming  $Y = \{x\}$ , we have

$$\delta[x := x + 2; \ x := x - 3] = \exists \tilde{x}: (\tilde{x} = x + 2 \wedge x' = \tilde{x} - 3),$$

which is equivalent to

$$x' = x - 1.$$

The definition of the data relation for  $[S_1; \dots; S_k]$  for  $k > 2$  is easily derivable from that of  $k = 2$ .

Let  $S$  be a composite statement that does not involve synchronous communication. With the grouped statement

$$\ell: \langle S \rangle; \hat{\ell}:$$

we associate a transition  $\tau_\ell$ , whose transition relation  $\rho_\ell$  is given by

$$\rho_\ell: move(\ell, \hat{\ell}) \wedge \delta[S].$$

Now, let us handle the case of grouped statements involving synchronous communications. For simplicity, we consider only a special case of grouped statements involving synchronous communications. Consider a pair of parallel grouped statements of the form

$$\ell: \langle T_1; \alpha \Leftarrow e; S_1 \rangle; \hat{\ell}: \quad \text{and} \quad m: \langle T_2; \alpha \Rightarrow u; S_2 \rangle; \hat{m}:$$

where  $T_1$ ,  $S_1$ ,  $T_2$ , and  $S_2$  do not contain any synchronous communication statements, and the variables referenced in  $T_1$ ,  $e$ , and  $S_1$  are disjoint from the variables referenced in  $T_2$ ,  $u$  ( $u$  itself), and in  $S_2$ . With each such pair we associate a joint transition  $\tau_{\langle \ell, m \rangle}$  with transition relation

$$\rho_{\langle \ell, m \rangle}: move(\{\ell, m\}, \{\hat{\ell}, \hat{m}\}) \wedge \delta[T_1; T_2; u := e; S_1; S_2].$$

The  $\delta$  conjunct of this formula implies a sequential execution of  $[T_1; T_2; u := e; S_1; S_2]$  which starts by performing  $T_1$  followed by  $T_2$ . It then assigns the value of  $e$  to variable  $u$ , and finally performs  $S_1$  followed by  $S_2$ . Under the assumptions made about  $T_1$ ,  $e$ ,  $S_1$ ,  $T_2$ ,  $u$ , and  $S_2$ , putting  $T_1$  ahead of  $T_2$  and  $S_1$  ahead of  $S_2$  in the definition of  $\rho_{\langle \ell, m \rangle}$  is completely arbitrary. Instead, we could have replaced  $\delta[T_1; T_2; u := e; S_1; S_2]$  by  $\delta[T_2; T_1; u := e; S_2; S_1]$  or by any permutation of  $T_1$ ,  $T_2$  followed by  $u := e$ , followed by any permutation of  $S_1$ ,  $S_2$ . Due to the noninterference of the two statements, except in the communication itself, all of these possibilities would lead to the same final effect.

Note that, except for the idling transition  $\tau_I$ , all transitions in the fair transition system corresponding to a program are self-disabling and cannot be taken twice in succession. This means that if  $s_2$  is a  $\tau$ -successor of  $s_1$ , for some non-idling transition  $\tau$ , then  $s_2$  has no  $\tau$ -successor for the same  $\tau$ . This property follows from the fact that each non-idling transition is associated with some location  $\ell$  and the corresponding transition relation includes  $at\_l$  as part of its enabling condition. When  $\tau$  is taken, control moves away from  $\ell$  to a different location in the same process, which causes  $\tau$  to become temporarily disabled.

## The Justice Set

Recall that a fair transition system includes two fairness components: a set  $\mathcal{J}$  of just transitions and a set  $\mathcal{C}$  of compassionate transitions. Placing a transition in either of these sets identifies the circumstances under which it must be taken.

The set of just transitions  $\mathcal{J}$  includes all transitions, excluding the idling transition  $\tau_i$  and the transitions associated with statements *noncritical* and *idle*.

This implies that control may stay in front of a *noncritical* or *idle* statement forever, representing the situation that execution of the noncritical section does not terminate. In contrast, the transition associated with a *critical* statement is in the justice set, implying that control cannot stay forever in front of a *critical* statement and, therefore, execution of a critical section always terminates.

## The Compassion Set

The set of compassionate transitions  $\mathcal{C}$  includes all transitions associated with the communication and semaphore statements

*send*, *receive*, and *request*.

In particular,  $\mathcal{C}$  includes the transitions for grouped statements that contain communication substatements.

The transitions placed in  $\mathcal{C}$  belong to statements that we wish to treat with extra fairness, because they represent competition between processes on a common resource, which is either a semaphore variable or a communication channel.

Observe that we do not require compassion for the *release* statement, because it represents the release of a resource for which there is no competition.

## Computations of Programs

A computation of a program is a computation of the fair transition system associated with the program.

To illustrate the consequences of the definitions, we consider several sample programs. Consider program SAMPLE1 of Fig. 0.4.

Process  $P_1$  of program SAMPLE1 has a selection statement at  $\ell_0$  that chooses between the execution of the *await* statement  $\ell_0^a$  and the *skip* statement  $\ell_0^b$ . Execution of any of these statements causes process  $P_1$  to reach location  $\ell_1$  and terminate. Thus, Process  $P_1$  has two transitions,  $\ell_0^a$  and  $\ell_0^b$ , while the transitions of process  $P_2$  are  $m_0$  and  $m_1$ .

An interesting question is whether process  $P_1$  terminates in all computations of program SAMPLE1. The answer to this question is affirmative. This is because

**local  $x$ : integer where  $x = 1$**

$$P_1 :: \left[ \begin{array}{l} \ell_0: \left[ \begin{array}{l} \ell_0^a: \text{await } x = 1 \\ \text{or} \\ \ell_0^b: \text{skip} \end{array} \right] \\ \ell_1: \end{array} \right] \quad || \quad P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ [m_1: x := -x] \end{array} \right]$$

Fig. 0.4. Program SAMPLE1.

the only candidate computation in which  $P_1$  does not terminate is the following:

$$\begin{aligned} \sigma_1: & \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \\ & \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \\ & \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \dots \end{aligned}$$

However, this sequence of states is unjust towards transition  $\ell_0^b$  which is enabled on all states but never taken. Consequently,  $\sigma_1$  is not a computation, and all the computations include a state in which  $P_1$  has terminated.

In comparison, consider program SAMPLE2 of Fig. 0.5, in which the skip statement at  $\ell_0^b$  is replaced by `await  $x \neq 1$` .

**local  $x$ : integer where  $x = 1$**

$$P_1 :: \left[ \begin{array}{l} \ell_0: \left[ \begin{array}{l} \ell_0^a: \text{await } x = 1 \\ \text{or} \\ \ell_0^b: \text{await } x \neq 1 \end{array} \right] \\ \ell_1: \end{array} \right] \quad || \quad P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ [m_1: x := -x] \end{array} \right]$$

Fig. 0.5. Program SAMPLE2.

This program has the following computation in which  $P_1$  does not terminate

$$\begin{aligned} \sigma_2: & \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \\ & \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \end{aligned}$$

$$\langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \dots$$

State sequence  $\sigma_2$  is a computation, even though  $\ell_0^a$  and  $\ell_0^b$  are not taken. This is because none of these just transitions is continuously enabled. Transition  $\ell_0^a$  is disabled on the infinitely many states at which  $x = -1$ , while  $\ell_0^b$  is disabled on the infinitely many states at which  $x = 1$ .

Finally, consider program SAMPLE3 of Fig. 0.6.

```

local x: integer where x = 1

P1 :: [ℓ₀: if x = 1 then
        ℓ₁: skip
        else
        ℓ₂: skip
        ℓ₃: ] || P₂ :: [m₀: loop forever do
                           [m₁: x := -x]
                           ]

```

Fig. 0.6. Program SAMPLE3.

This program replaces the selection statement of program SAMPLE2 by a conditional statement. While the conditional statement of SAMPLE3 appears to perform a similar action to the selection statement in SAMPLE2, it differs from the selection statement in having a single transition instead of the two transitions associated with the selection. Consequently, state sequence  $\sigma_2$  is not a computation of program SAMPLE3 since statement  $\ell_0$  is enabled on all states of  $\sigma_2$ . It follows that, similar to program SAMPLE1, process  $P_1$  terminates in all computations of SAMPLE3.

In Problem 0.2, the reader is requested to consider a program and draw conclusions about the behavior of its computations. Problem 0.3 introduces the notion of congruence between statements and requests the reader to identify some congruent statements.

## The Mutual-Exclusion Problem

A typical process in a concurrent system has some activities that it can perform with no need for coordination with other processes, while other activities may require such coordination. One of the cases in which coordination is needed is when there are some resources that must be used exclusively, i.e., by one process at a time, and these resources are shared between processes. For example, to

print a long document on a printer, the printing task (process) requires uninterrupted possession of the printer. It is customary to use the schematic statement **noncritical** to represent the uncoordinated activity, which is referred to as the *noncritical section*, while the statement **critical** represents the activity that requires coordination, called the *critical section*.

The *mutual-exclusion problem* for two processes is to devise a protocol (program) consisting of two processes. Each process should contain the two schematic **noncritical** and **critical** statements as well as some coordination statements separating them. The role of the coordination statements is to guarantee exclusive execution of the critical sections, i.e., while one of the processes is in its critical statement, the other is not.

Program MUX-SEM of Fig. 0.7 presents one of the classical solutions to the mutual-exclusion problem, where coordination is achieved by semaphores.

local  $y$ : integer where  $y = 1$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } y \\ \ell_3: \text{critical} \\ \ell_4: \text{release } y \end{array} \right] \end{array} \right]$	$\parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \text{request } y \\ m_3: \text{critical} \\ m_4: \text{release } y \end{array} \right] \end{array} \right]$
--	---

Fig. 0.7. Program MUX-SEM (mutual exclusion by semaphores).

A good solution to the mutual-exclusion problem is expected to satisfy the following two requirements:

- *Exclusion:* No computation of the program should include a state in which both processes are executing in their critical sections at the same time (e.g.,  $P_1$  is at location  $\ell_3$ , while  $P_2$  is at location  $m_3$  in program MUX-SEM).
- *Accessibility:* Every state in a computation in which one process is at the noncritical section exit (e.g., location  $\ell_2$  for  $P_1$  in MUX-SEM) must be eventually followed by another state in which the process is in its critical section (e.g.,  $\ell_3$  for  $P_1$  in MUX-SEM).

We show that the requirements of exclusion and accessibility are satisfied by program MUX-SEM.

Assume, for example, that  $P_1$  arrives first at  $\ell_2$  when  $y$  is 1. It proceeds beyond  $\ell_2$  while setting  $y$  to 0. As long as  $P_1$  is at  $\ell_3$  or  $\ell_4$ ,  $y$  remains 0. Con-

sequently, if the other process  $P_2$  attempts to proceed beyond its **request**  $y$  statement at  $m_2$ , it will be suspended there since the enabling condition  $y > 0$  is false. Process  $P_2$  must therefore wait for  $y$  to turn positive, which can only be caused by  $P_1$  performing **release**  $y$  at  $\ell_4$ . Similarly, when  $P_2$  is anywhere at  $m_3$  or  $m_4$ ,  $y$  is 0, and  $P_1$  is then barred from entering its critical section.

This shows that program MUX-SEM satisfies the requirement of mutual exclusion.

To show that the program also satisfies the requirement of accessibility, assume by contradiction that process  $P_1$  gets stuck at  $\ell_2$  at some point of the computation, and never reaches  $\ell_3$  beyond this point. If  $y$  remains continuously positive beyond some point, then the transition of  $\ell_2$  is infinitely often enabled (in fact, it is continuously enabled). If  $y$  does not remain continuously positive, this can happen only as a result of an eventually periodic computation of the following form (we list for each state the values of the variables  $\pi$  and  $y$ ):

$$\begin{aligned} \sigma: \quad & \langle \{\ell_0, m_0\}, 1 \rangle \longrightarrow \quad \longrightarrow \boxed{\langle \{\ell_2, m_2\}, 1 \rangle} \xrightarrow{m_2} \\ & \langle \{\ell_2, m_3\}, 0 \rangle \xrightarrow{m_3} \langle \{\ell_2, m_3\}, 0 \rangle \xrightarrow{m_4} \langle \{\ell_2, m_0\}, 1 \rangle \xrightarrow{m_0} \\ & \langle \{\ell_2, m_1\}, 1 \rangle \xrightarrow{m_1} \boxed{\langle \{\ell_2, m_2\}, 1 \rangle} \xrightarrow{m_2} \langle \{\ell_2, m_3\}, 0 \rangle \longrightarrow \dots . \end{aligned}$$

However, this computation visits infinitely many times the state  $\langle \{\ell_2, m_2\}, 1 \rangle$ , at which transition  $\ell_2$  is enabled. It follows that, regardless of whether  $y$  remains continuously positive or not, transition  $\ell_2$  is enabled infinitely many times beyond the point that  $P_1$  got stuck. Since  $\ell_2$ , being associated with a **request** statement, is a compassionate transition, it must eventually be taken, contradicting the assumption that  $P_1$  got stuck at  $\ell_2$ . This also shows that the sequence of states  $\sigma$  is not a computation since it violates the requirement of compassion for  $\ell_2$ .

The solution of the mutual-exclusion problem by semaphores easily generalizes to the case of several processes.

### Communal Accessibility

In Fig. 0.8, we present program MUX-WHEN which may also be proposed as a solution to the mutual-exclusion problem.

This program can be obtained from program MUX-SEM by replacing the statement **request**  $y$  by the grouped statement **(when**  $y > 0$  **do**  $y := y - 1$ ) and replacing the statement **release**  $y$  by  $y := y + 1$ . For both replacements, the transition relation of the original statement is equal to that of the replacing statement. The only difference is in the fairness assumptions associated with the two statements. The transition associated with the **request**  $y$  statement is compassionate, while the transition associated with **(when**  $y > 0$  **do**  $y := y - 1$ ) is only required to be just.

```

local y: integer where y = 1

P1 :: [l0: loop forever do
        [l1: noncritical
         l2: <when y > 0 do y := y - 1>
         l3: critical
         l4: y := y + 1]]]
      ||
P2 :: [m0: loop forever do
        [m1: noncritical
         m2: <when y > 0 do y := y - 1>
         m3: critical
         m4: y := y + 1]]]

```

Fig. 0.8. Program MUX-WHEN (mutual exclusion by grouped statements).

Arguments similar to the ones used for program MUX-SEM show that program MUX-WHEN also satisfies the requirement of mutual exclusion. However, it does not satisfy the property of accessibility. In fact, the sequence of states  $\sigma$  presented previously and shown not to be a computation of MUX-SEM is a computation of program MUX-WHEN. In this computation, process  $P_2$  is stuck forever at location  $l_2$  and never reaches  $l_3$ . This demonstrates the importance of the compassion requirements in ensuring the property of accessibility.

While program MUX-SEM does not satisfy the requirement of accessibility, it satisfies a weaker notion which can be defined as follows:

- *Communal accessibility:* Every state in a computation in which some process is at the noncritical section exit (e.g., location  $l_2$  for process  $P_1$  in program MUX-WHEN) must be eventually followed by another state in which some process (not necessarily the same) is in its critical section (e.g.,  $l_3$  for  $P_1$  or  $m_3$  for  $P_2$  in MUX-WHEN).

Obviously, program MUX-WHEN satisfies the requirement of communal accessibility, since all computations in which an  $l_2$ -state is not followed by an  $l_3$ -state, such as the previously considered  $\sigma$ , are such that every  $l_2$ -state is followed by an  $m_3$ -state.

In Problem 0.4, the reader is requested to consider two proposed solutions

to the mutual-exclusion problem, and analyze their properties.

## Control Configurations

As previously stated, the control variable  $\pi$  ranges over sets of locations. However, not every set of locations can appear as the value of  $\pi$  in a computation.

Consider a program  $P$ . A set of locations  $L = \{[\ell_1], \dots, [\ell_k]\}$  of  $P$  is called *conflict-free* if no  $[\ell_i]$  conflicts with  $[\ell_j]$ , for  $i \neq j$ . A set of locations  $L$  is called a (*control*) *configuration* of  $P$  if it is a maximal conflict-free set, i.e.,  $L$  is conflict-free but any  $L'$ ,  $L \subsetneq L'$ , contains some conflicting locations.

Consider, for example, the program whose body is:

$$[\ell_1: S_1; \quad \ell_2: \left[ \begin{array}{l} \ell_3: S_3 \\ \ell_4: \end{array} \right] \parallel \left[ \begin{array}{l} \ell_5: S_5 \\ \ell_6: \end{array} \right]; \quad \ell_7: [\text{while } c \text{ do } \ell_8: S_8]; \quad \ell_9:]$$

where  $S_1$ ,  $S_3$ ,  $S_5$ , and  $S_8$  are assignment statements. The set of configurations for this program consists of

$$\begin{aligned} &\{[\ell_1]\}, \{[\ell_2]\}, \{[\ell_3], [\ell_5]\}, \{[\ell_4], [\ell_5]\}, \{[\ell_3], [\ell_6]\}, \\ &\{[\ell_4], [\ell_6]\}, \{[\ell_7]\}, \{[\ell_8]\}, \{[\ell_9]\}. \end{aligned}$$

It is not difficult to see that any value assumed by the control variable  $\pi$  is a control configuration. Thus, the set of control configurations is the domain over which variable  $\pi$  ranges. However, not every control configuration can be assumed as a value of  $\pi$ .

Consider, for example, the following program:

$$\begin{aligned} &\text{local } x: \text{integer where } x = 0 \\ P_1 :: &\left[ \begin{array}{l} \ell_0: x := 1 \\ \ell_1: \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: \text{await } x = 1 \\ m_1: \end{array} \right] \end{aligned}$$

The configurations of this program are

$$\{[\ell_0], [m_0]\}, \quad \{[\ell_0], [m_1]\}, \quad \{[\ell_1], [m_0]\}, \quad \{[\ell_1], [m_1]\}.$$

Of these, the configuration  $\{[\ell_0], [m_1]\}$  does not appear in any accessible state. This is because  $P_2$  cannot move to  $m_1$  before  $P_1$  sets  $x$  to 1 and moves to  $\ell_1$ .

A configuration that appears as the value of  $\pi$  in some accessible state is called an *accessible configuration*.

Clearly, for a given program  $P$ , the set of configurations is always finite and can be effectively constructed. The question of whether a given configuration is accessible can be shown to be undecidable.

## Examples of Message-Passing Programs

To illustrate communication by message passing, we present examples of two programs.

### Example (producer-consumer)

In Fig. 0.9 we present program PROD-CONS, which models a producer-consumer situation for a fixed  $N > 0$ .

**local** *send, ack*: channel [1..] of integer  
**where** *send* =  $\Lambda$ , *ack* =  $\underbrace{[1, \dots, 1]}_N$

*Prod* ::  $\left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \text{ack} \Rightarrow t \\ \ell_3: \text{send} \Leftarrow x \end{array} \right] \end{array} \right] \parallel \text{Cons} :: \left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{send} \Rightarrow y \\ m_2: \text{ack} \Leftarrow 1 \\ m_3: \text{consume } y \end{array} \right] \end{array} \right]$

Fig. 0.9. Program PROD-CONS (producer-consumer).

The producer process *Prod* computes a value and places it in variable *x*. The details of the computation are irrelevant, and we use the schematic statement **produce** *x* to represent the production activity. After production, process *Prod* uses asynchronous channel *send* to transmit at  $\ell_3$  the value *x* to the consumer process *Cons*. Part of the problem formulation is that channel *send* should never hold more than  $N$  values at the same time. To guarantee this requirement, the program uses asynchronous channel *ack* which serves as an acknowledgement channel. Each message in channel *ack* can be viewed as a permission to send one message through *send*. The particular value of messages in *ack* is of no importance and we use the value 1. In accordance with this interpretation of *ack* messages, the producer removes at  $\ell_2$  one “permission” from *ack* before it sends the value of *x* onto *send*.

The initial value of channel *ack* is a sequence of  $N$  messages with value 1, implying  $N$  “permissions.” The cycle of the consumer consists of reading a value from channel *send*, placing it in *y*, and then using it in the **consume** *y* statement at  $m_3$ . In addition, it takes care to replenish the stock of permissions in *ack* by adding a new permission for each message taken off *send*.

**Example** (a fair merge by synchronous communication)

In Fig. 0.10, we present program FAIR-MERGE which merges two input streams generated by two producers into a single output stream in a fair manner.

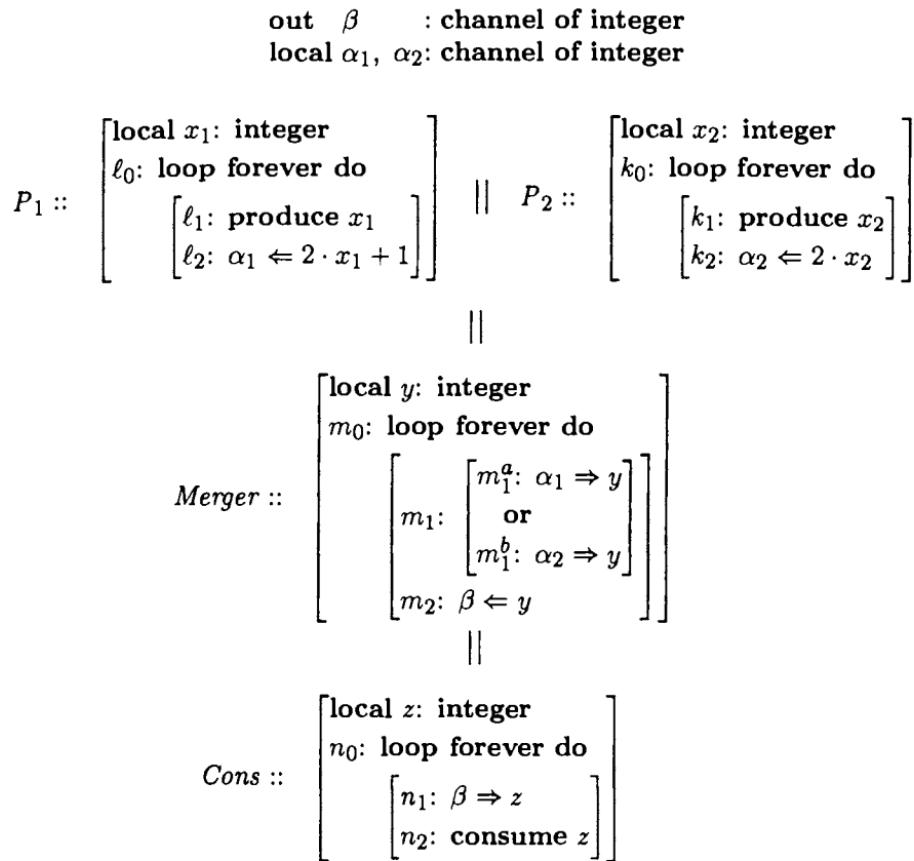


Fig. 0.10. Program FAIR-MERGE.

The program consists of two producer processes  $P_1$  and  $P_2$ , a single merger process  $Merger$ , and a single consumer process  $Cons$ . Each producer  $P_i$ ,  $i = 1, 2$ , alternates between a production of a value in  $x_i$  and transmission on channel  $\alpha_i$  of the value of an expression that depends on  $x_i$ . To distinguish between the values transmitted by the different producers, it is arranged that  $P_1$  sends odd values while  $P_2$  sends even values.

Process  $Merger$  has a nondeterministic selection at  $m_1$  in which it selects

to read a value from channel  $\alpha_1$  or channel  $\alpha_2$  into  $y$ . At  $m_2$  the value of  $y$  is transmitted on output channel  $\beta$ , to be read and consumed by process *Cons*.

Obviously, the sequence of values transmitted on channel  $\beta$  is some mixture of the values transmitted by the two producers. Due to the compassion requirements associated with communication statements, this mixture is *fair*. That is, if the sequence of values transmitted on  $\beta$  is infinite, it must contain infinitely many odd values (sent by  $P_1$ ) and infinitely many even values (sent by  $P_2$ ).

We argue that the mixture must be fair. A potential behavior in which the mixture is unfair can be generated only if, from a certain point on, one of the producers, say  $P_1$ , is stuck at its sending location ( $\ell_2$  for process  $P_1$ ) and process *Merger* consistently prefers communicating with its competitor (i.e., taking  $m_1^b$  rather than  $m_1^a$ ). Such a behavior generates an infinite sequence of states at which  $P_1$  is at  $\ell_2$  while *Merger* is at  $m_1$ . Hence, the synchronous communication transition  $(\ell_2, m_1^a)$  is enabled infinitely many times but never taken beyond the point at which  $P_1$  gets stuck at  $\ell_2$ . Such a behavior violates the requirement of compassion with respect to the compassionate transition  $(\ell_2, m_1^a)$  and cannot, therefore, be a computation. It follows that, in any computation, *Merger* must read infinitely many times from channel  $\alpha_1$  and infinitely many times from channel  $\alpha_2$ . ■

In **Problem 0.5**, the reader is requested to consider several message-passing solutions to the mutual-exclusion problem.

## 0.4 Modules

The preceding discussion showed how programs can be interpreted as fair transition systems. Most of the examples presented in this book concern complete programs.

However, with a small extension to the syntax, it is possible to use the previously introduced programming language to describe *modules* and interpret them as fair transition systems.

Both programs and modules are programming units with identical structure, consisting of a declaration followed by a body which is a cooperation statement. The difference between a program and a module is in the assumptions about the interaction between the unit and its environment. For a program, we assume that the environment may have set the initial values for the input variables but, once the program starts running, the environment cannot modify any of the system variables. Consequently, all changes to the system variables are accounted for by the transitions of the program. We can say that programs are *closed systems*, not interacting with external agents while they run.

In contrast, while a module is running, the environment may independently change some of the system variables. Thus, a module should be viewed as an *open system* that can interact with its environment while it is running. We will discuss below which variables can be modified by the environment.

We may consider a module as a process in a bigger program, whose details are not available to us. According to this view, environmental changes to variables known to the module are effected by one of the other (unknown) processes, running in parallel to the module.

Note that open systems, as represented by modules, capture the essence of reactive systems, where interaction with the environment is an integral part of their behavior.

## Syntax of Modules

A module has the form

$$\begin{aligned} &\text{module module-name} \\ &\text{module-declaration} \\ &[P_1 :: [\ell_1: S_1; \hat{\ell}_1: ] \parallel \cdots \parallel P_k :: [\ell_k: S_k; \hat{\ell}_k: ]] \end{aligned}$$

The keyword **module** identifies the unit as a module rather than a program.

*Module-declaration* is a list of module-declaration statements that are the same as declaration statements for a program, except that they allow two additional modes:

**own in** — Specifies asynchronous channels that can be read only by the module. This mode guarantees that, once a message is appended to such a channel by the environment, it can be removed only by the module.

**own out** — Specifies variables or asynchronous channels that can be written only by the module, but may be read by the environment.

For simplicity, we exclude synchronous communication statements from modules.

**Example** In Fig. 0.11, we present a module that acts as a buffer. The module communicates with the environment via asynchronous input channel  $\alpha$  and asynchronous output channel  $\beta$ .

The body of the module consists of two processes. Process *In* reads messages from channel  $\alpha$  and appends them to the end of local list  $b$ . Process *Out* removes messages from the top of  $b$  and sends them on channel  $\beta$ . Channel  $\alpha$  is owned by **BUFFER** for input, while channel  $\beta$  is owned by **BUFFER** for output. The environment cannot modify any of the local variables  $x$ ,  $y$ , or  $b$ . It can only add messages

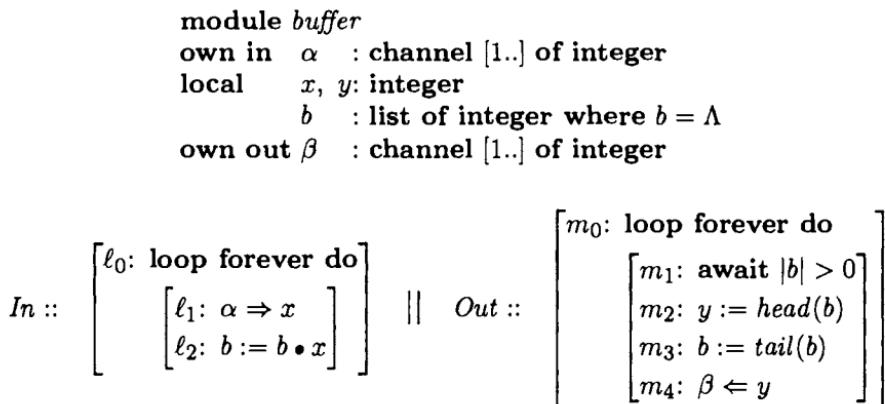


Fig. 0.11. Module BUFFER.

to the end of  $\alpha$  and remove messages from the front of  $\beta$ . ■

## Semantics of Modules

It is straightforward to construct a fair transition system  $S_M$  that represents the possible behaviors of a module  $M$ . The system variables, initial condition, justice set, and compassion set of  $S_M$  are the same as if  $M$  were a program. As the set of transitions  $T_M$  we take the idling transition  $\tau_I$ , all the transitions associated with the statements of  $M$ 's body, and a new transition  $\tau_E$  called the *environment transition*. Transition  $\tau_E$  is intended to represent all the changes the environment may apply to the system variables.

The transition relation  $\rho_{\tau_E}$  is given by

$$\rho_{\tau_E}: \bigwedge_{u \in V} env(u),$$

where  $env(u)$  describes the possible modifications of  $u$  by the environment. The formula  $env(u)$  is defined as follows:

- For the control variable  $\pi$ ,

$$env(\pi): \bigwedge_{\ell \in L_M} (at'_\ell \leftrightarrow at_\ell),$$

where  $L_M$  is the set of locations in module  $M$ , and the symbol  $\leftrightarrow$  denotes equivalence. Thus, the environment may arbitrarily modify  $\pi$  provided it preserves the control locations of  $M$ .

- If  $u$  is a variable that is specified to be of mode *local* or mode *own out* then  
 $\text{env}(u): u' = u.$

Thus, variables belonging to the described class cannot be modified by the environment.

- If  $\alpha$  is an asynchronous channel of mode *local*, then  
 $\text{env}(\alpha): \alpha' = \alpha.$

Thus, local asynchronous channels are also preserved by any environment action.

- If  $\alpha$  is an asynchronous channel of mode *own in*, then

$$\text{env}(\alpha): \alpha' = \alpha \vee \exists v: \alpha' = \alpha \bullet v.$$

This formula states that  $\alpha$  can only change by having a message  $v$  appended to its end. Thus, the environment may not remove any message from  $\alpha$ .

- If  $\alpha$  is an asynchronous channel of mode *own out*, then

$$\text{env}(\alpha): \alpha' = \alpha \vee \exists v: \alpha = v \bullet \alpha'.$$

This formula states that  $\alpha'$  can only differ from  $\alpha$  by removing a message  $v$  from the front of  $\alpha$  to obtain  $\alpha'$ . Thus, the environment cannot add a new message to  $\alpha$ .

- For all other variables and asynchronous channels  $u$

$$\text{env}(u): \top.$$

This formula allows  $u$  to change arbitrarily due to an environment transition.

## Processes as Modules

The fact that we can isolate a process from a program, call it a module, and verify its properties independently of the rest of the program has far-reaching consequences. In fact, the only hope for verifying a large program is by decomposing it into many small modules, and establishing properties of each individual module while considering the rest of the processes as its “environment.” We then infer properties of the entire program from the properties of its components. This style of verification is called *compositional verification*.

Let  $P_i$  be a process within a program  $P$ . As mentioned above, it is always possible to view  $P_i$  as a module and the rest of the program as its environment. The necessary transformation involves movement of declaration statements from the program heading to the module’s heading and identification of the variables that serve as inputs, outputs, and locals to the module, and the variables that are owned by the module.

**local  $x, y$ : integer where  $x = y = 0$**

$$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{await } x < y + 1 \\ \ell_2: x := x + 1 \end{array} \right] \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{await } y < x + 1 \\ m_2: y := y + 1 \end{array} \right] \end{array} \right]$$

Fig. 0.12. Program KEEPING-UP.

Consider, for example, the program KEEPING-UP presented in Fig. 0.12.

Process  $P_1$  in this program repeatedly increments  $x$ , provided  $x$  does not exceed  $y + 1$ . In a symmetric way, process  $P_2$  repeatedly increments  $y$ , provided  $y$  does not exceed  $x + 1$ .

We can view processes  $P_1$  and  $P_2$  as modules. The corresponding modules are presented in Fig. 0.13 and Fig. 0.14, respectively.

```
module M1
in      y : integer where y = 0
own out x : integer where x = 0

ℓ₀: loop forever do
    [ℓ₁: await x < y + 1
     ℓ₂: x := x + 1]
```

Fig. 0.13. Module  $M_1$  corresponding to process  $P_1$ .

Let  $M_i$  be the module corresponding to process  $P_i$  in a program  $P$ . We refer to computations of  $M_i$  as the *modular computations* of  $P_i$ , i.e., the computations obtained when we view  $P_i$  as a module (with appropriate identification of the modes of the variables). For example, the following state sequence is a prefix of a modular computation of process  $P_1$  in program KEEPING-UP even though it is not a prefix of any computation of this program:

```

module M2
in      x : integer where x = 0
own out y : integer where y = 0

m0: loop forever do
    
$$\begin{bmatrix} m_1: \text{await } y < x + 1 \\ m_2: y := y + 1 \end{bmatrix}$$


```

Fig. 0.14. Module  $M_2$  corresponding to process  $P_2$ .

$$\begin{aligned} \sigma: \langle \pi: \{\ell_0, m_0\}, x: 0, y: 0 \rangle &\xrightarrow{\ell_0} \langle \pi: \{\ell_1, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\ &\langle \pi: \{\ell_2, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\tau_E} \langle \pi: \{\ell_2, m_0\}, x: 0, y: -1 \rangle \xrightarrow{\ell_2} \\ &\langle \pi: \{\ell_0, m_0\}, x: 1, y: -1 \rangle. \end{aligned}$$

The following claim establishes a connection between computations of the entire program and modular computations of its processes.

**Claim** Every computation of a program is a modular computation of each of its top-level processes.

Thus, the set of computations of the entire program is a subset of the set of modular computations of each of its top-level processes. The restriction to top-level processes is necessary to avoid the case of a process terminating and restarting again, which is not covered by modular computations of a process.

**Example** Consider, for example, the following computation of program KEEPING-UP

$$\begin{aligned} \sigma: \langle \pi: \{\ell_0, m_0\}, x: 0, y: 0 \rangle &\xrightarrow{\ell_0} \langle \pi: \{\ell_1, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\ &\langle \pi: \{\ell_2, m_0\}, x: 0, y: 0 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_2, m_1\}, x: 0, y: 0 \rangle \xrightarrow{\ell_2} \\ &\langle \pi: \{\ell_0, m_1\}, x: 1, y: 0 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_2\}, x: 1, y: 0 \rangle \xrightarrow{m_2} \\ &\langle \pi: \{\ell_0, m_0\}, x: 1, y: 1 \rangle \xrightarrow{\ell_0} \langle \pi: \{\ell_1, m_0\}, x: 1, y: 1 \rangle \dots \end{aligned}$$

Viewed as a modular computation of process  $P_1$ , this computation can be presented as:

$$\sigma_1: \langle \pi: \{\ell_0, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\ell_0} \langle \pi: \{\ell_1, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\ell_1} \\ \langle \pi: \{\ell_2, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\tau_E} \langle \pi: \{\ell_2, m_1\}, x: 0, y: 0 \rangle \xrightarrow{\ell_2} \\ \langle \pi: \{\ell_0, m_1\}, x: 1, y: 0 \rangle \xrightarrow{\tau_E} \langle \pi: \{\ell_0, m_2\}, x: 1, y: 0 \rangle \xrightarrow{\tau_E} \\ \langle \pi: \{\ell_0, m_0\}, x: 1, y: 1 \rangle \xrightarrow{\ell_0} \langle \pi: \{\ell_1, m_0\}, x: 1, y: 1 \rangle \dots .$$

Viewed as a modular computation of process  $P_2$ , this computation can be presented as:

$$\sigma_2: \langle \pi: \{\ell_0, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\tau_E} \langle \pi: \{\ell_1, m_0\}, x: 0, y: 0 \rangle \xrightarrow{\tau_E} \\ \langle \pi: \{\ell_2, m_0\}, x: 0, y: 0 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_2, m_1\}, x: 0, y: 0 \rangle \xrightarrow{\tau_E} \\ \langle \pi: \{\ell_0, m_1\}, x: 1, y: 0 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_2\}, x: 1, y: 0 \rangle \xrightarrow{m_2} \\ \langle \pi: \{\ell_0, m_0\}, x: 1, y: 1 \rangle \xrightarrow{\tau_E} \langle \pi: \{\ell_1, m_0\}, x: 1, y: 1 \rangle \dots .$$

This illustrates that a computation of a program is a modular computation of each of its top-level processes. ■

## 0.5 Temporal Logic

In this section, we introduce the language of *temporal logic*, which is our language of choice for specifying properties of reactive systems. Temporal logic extends ordinary predicate logic by introducing a set of special temporal operators that provide a natural, succinct, and abstract description of precedence, invariance, and frequent recurrence of events in time.

We assume an underlying assertion language  $\mathcal{L}$  which is a first-order language over interpreted symbols for expressing functions and relations over some concrete domains such as integers, arrays, and lists of integers.

We refer to a formula in the assertion language  $\mathcal{L}$  as a *state formula*, or simply as an *assertion*. For an assertion  $p$  and a state  $s$  that interprets all variables appearing in  $p$ , we write

$$s \models p$$

to denote that  $s$  *satisfies*  $p$ , which can also be described by saying that  $p$  *holds* on  $s$ , or that  $s$  is a  $p$ -*state*.

A *temporal formula* (a *formula*) is constructed out of state formulas to which we apply the boolean connectives  $\neg$  (negation) and  $\vee$  (disjunction), quantifiers  $\forall$  and  $\exists$ , and the temporal operators presented in the table of Fig. 0.15.

Future Temporal Operators		Past Temporal Operators	
$\Box p$	- Henceforth $p$	$\Xi p$	- So-far $p$
$\Diamond p$	- Eventually $p$	$\Diamond p$	- Once $p$
$p \mathcal{U} q$	- $p$ Until $q$	$p \mathcal{S} q$	- $p$ Since $q$
$p \mathcal{W} q$	- $p$ Waiting-for (Unless) $q$	$p \mathcal{B} q$	- $p$ Back-to $q$
$\bigcirc p$	- Next $p$	$\ominus p$	- Previously $p$
		$\oslash p$	- Before $p$

Fig. 0.15. The temporal operators.

The other boolean connectives, such as  $\wedge$  (conjunction),  $\rightarrow$  (implication), and  $\leftrightarrow$  (equivalence), can be defined in terms of  $\neg$  and  $\vee$ .

When writing temporal formulas that are not fully parenthesized, temporal operators have higher binding power than the boolean ones. Thus,  $p \mathcal{U} q \vee u \mathcal{W} v$  is interpreted as  $(p \mathcal{U} q) \vee (u \mathcal{W} v)$ .

## Semantics of Temporal Formulas

Temporal formulas are interpreted over a *model*, which is an infinite sequence of states  $\sigma: s_0, s_1, \dots$ . Note that all computations are models. Variables of the vocabulary  $\mathcal{V}$  are partitioned into flexible and rigid variables. *Flexible variables* may assume different values in different states, while *rigid variables* must assume the same value in all states of the model. System variables are always flexible.

Given a model  $\sigma$  and a temporal formula  $p$ , we present an inductive definition for the notion of  $p$  holding at a position  $j \geq 0$  in  $\sigma$ , denoted by

$$(\sigma, j) \models p.$$

- For a state formula  $p$ ,

$$(\sigma, j) \models p \iff s_j \models p.$$

That is, we evaluate  $p$  locally, using the interpretation given by  $s_j$ .

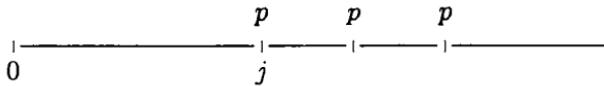
For the boolean connectives,

- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$ , i.e., not  $(\sigma, j) \models p$ .
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$  or  $(\sigma, j) \models q$ .

For the future operators,

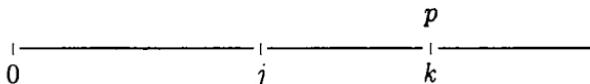
- $(\sigma, j) \models \Box p \iff$  for all  $k \geq j$ ,  $(\sigma, k) \models p$ .

Thus,  $\Box p$  holds at position  $j$  iff  $p$  holds at position  $j$  and all following positions. Pictorially:



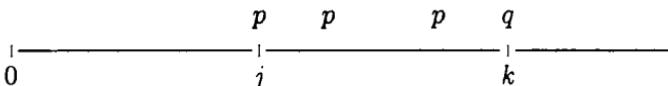
- $(\sigma, j) \models \Diamond p \iff \text{for some } k \geq j, (\sigma, k) \models p.$

Thus,  $\Diamond p$  holds at position  $j$  iff  $p$  holds at some position  $k \geq j$ .



- $(\sigma, j) \models p \cup q \iff \text{for some } k \geq j, (\sigma, k) \models q,$   
and for every  $i$  such that  $j \leq i < k$ ,  $(\sigma, i) \models p$ .

Thus,  $p \cup q$  holds at position  $j$  iff  $q$  holds at some position  $k \geq j$  and  $p$  holds continually from  $j$  through  $k - 1$ .

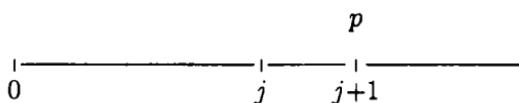


- $(\sigma, j) \models p \mathcal{W} q \iff (\sigma, j) \models p \cup q \text{ or } (\sigma, j) \models \Box p.$

That is, waiting-for is a weak version of the until operator which allows the possibility that  $q$  never occurs and, in this case,  $p$  holds at  $j$  and all positions beyond  $j$ .

- $(\sigma, j) \models \bigcirc p \iff (\sigma, j + 1) \models p.$

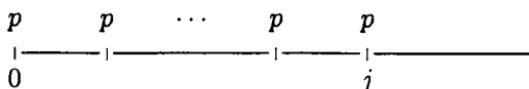
Thus,  $\bigcirc p$  holds at position  $j$  iff  $p$  holds at the next position  $j + 1$ .



For the past operators,

- $(\sigma, j) \models \Box p \iff \text{for all } k, 0 \leq k \leq j, (\sigma, k) \models p.$

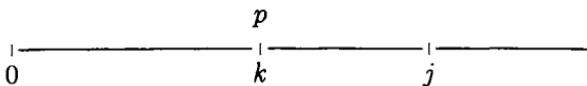
Thus,  $\Box p$  holds at position  $j$  iff  $p$  holds at  $j$  and all preceding positions.



- $(\sigma, j) \models \Diamond p \iff \text{for some } k, 0 \leq k \leq j, (\sigma, k) \models p.$

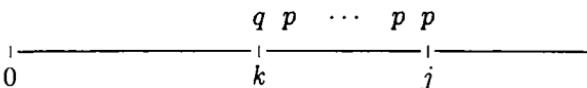
Thus,  $\Diamond p$  holds at position  $j$  iff  $p$  holds at  $j$  or at some preceding position

$k, 0 \leq k \leq j$ .



- $(\sigma, j) \models p \mathcal{S} q \iff$  for some  $k, 0 \leq k \leq j$ ,  $(\sigma, k) \models q$ ,  
and for every  $i$  such that  $k < i \leq j$ ,  $(\sigma, i) \models p$ .

Thus,  $p \mathcal{S} q$  holds at position  $j$  iff  $q$  holds at some position  $k \leq j$  and  $p$  holds continually from  $k + 1$  to  $j$ .

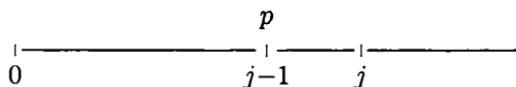


- $(\sigma, j) \models p \mathcal{B} q \iff (\sigma, j) \models p \mathcal{S} q$  or  $(\sigma, j) \models \Box p$

That is, back-to is a weak version of the since operator which allows the possibility that  $q$  never occurs and, in this case,  $p$  holds at  $j$  and all positions preceding  $j$ .

- $(\sigma, j) \models \ominus p \iff j > 0$  and  $(\sigma, j - 1) \models p$ .

Thus,  $\ominus p$  holds at position  $j$  iff  $j$  is not the first position in the model and  $p$  holds at position  $j - 1$ .



- $(\sigma, j) \models \odot p \iff$  either  $j = 0$  or else  $(\sigma, j) \models \ominus p$ .

Thus,  $\odot$  holds at position  $j$  iff either  $j$  is the first position in the model or  $p$  holds at position  $j - 1 \geq 0$ .

Let  $\sigma: s_0, s_1, \dots$ , and  $\sigma': s'_0, s'_1, \dots$  be two models, and  $u$  a (flexible or rigid) variable. Recall (page 4) that  $\sigma'$  is a  $u$ -variant of  $\sigma$  if for each  $j \geq 0$ ,  $s'_j$  differs from  $s_j$  by at most the interpretation given to  $u$ .

For the quantifiers,

- $(\sigma, j) \models \exists u: p \iff (\sigma', j) \models p$  for some  $\sigma'$ , a  $u$ -variant of  $\sigma$ ,
- $(\sigma, j) \models \forall u: p \iff (\sigma', j) \models p$  for every  $\sigma'$ , a  $u$ -variant of  $\sigma$ .

This concludes the definition of  $(\sigma, j) \models p$ .

For a (temporal) formula  $p$  and a position  $j \geq 0$  such that  $(\sigma, j) \models p$ , we say that  $p$  holds at position  $j$  of  $\sigma$  and also that  $j$  is a  $p$ -position (in  $\sigma$ ). Note that the satisfaction of a past formula at position  $j \geq 0$  depends only on the finite prefix  $s_0, \dots, s_j$ .

If  $(\sigma, 0) \models p$ , we say that  $p$  holds on  $\sigma$ , and denote it by

$$\sigma \models p.$$

We refer to  $\square$ ,  $\diamond$ ,  $\mathcal{U}$ ,  $\mathcal{W}$ , and  $\bigcirc$  as *future operators* and to  $\Box$ ,  $\Diamond$ ,  $\mathcal{S}$ ,  $\mathcal{B}$ ,  $\ominus$ , and  $\oslash$  as *past operators*. A formula that contains no future operators is called a *past formula*. A formula that contains no past operators is called a *future formula*. Note that a state formula (assertion) is both a past and a future formula.

A useful past formula is the formula *first*, defined as

$$\text{first: } \neg \ominus \top.$$

This formula characterizes the first position in any model. That is, it is false at all positions  $j > 0$  and true for  $j = 0$ .

**Examples** Following are several temporal formulas and their intended meaning. The verbal description characterizes the models  $\sigma$  such that  $\sigma \models \varphi$  for the considered formula  $\varphi$ , i.e., that  $\varphi$  holds at position 0 of  $\sigma$ . We assume that the subformulas  $p$ ,  $q$ , and  $r$  appearing below are state formulas.

- $p \rightarrow \diamond q$

This formula states that if the model  $\sigma$  satisfies  $p$  it also satisfies  $\diamond q$ . A model  $\sigma$  satisfies  $p$  if  $p$  is true in  $s_0$ . It satisfies  $\diamond q$ , if for some position  $j \geq 0$ ,  $q$  holds at  $s_j$ . Consequently, this formula expresses the following property of the model  $\sigma$ :

if initially  $p$  then eventually  $q$ .

- $\Box(p \rightarrow \diamond q)$

The previous formula stated the property: "if  $p$  holds at position  $j$  then  $q$  holds at some position not smaller than  $j$ " for  $j = 0$ . Adding the henceforth operator in front of the previous formula states that this property holds for all positions  $j \geq 0$ . Consequently, this formula expresses the following property of  $\sigma$ :

every  $p$ -position coincides with or is followed by a  $q$ -position.

- $\Box \diamond q$

This formula can be obtained from the previous formula by omitting the  $p$  condition (i.e., taking  $p = \top$ ). We therefore obtain a property that states that every position in the sequence coincides with or is followed by a later position satisfying  $q$ . Consequently, the formula states that

the sequence  $\sigma$  contains infinitely many  $q$ -positions.

- $\diamond \Box q$

Reading the formula from left to right, we obtain the following: there exists a position  $j \geq 0$  that satisfies  $\Box q$ , i.e., there exists a position such that  $q$

holds at all later positions. This property can be described as the following property of  $\sigma$ :

eventually permanently  $q$ ,

or equivalently:

the sequence  $\sigma$  contains only finitely many  $\neg q$ -positions.

- $(\neg q) \mathcal{W} p$

The formula states that either  $\neg q$  holds forever or that it holds until an occurrence of  $p$ . This means:

the first  $q$ -position must coincide with or be preceded by a  $p$ -position.

- $\square(p \rightarrow q \mathcal{W} r)$

This formula states that every  $p$ -position initiates an interval of continuous  $q$  which either extends to infinity or is terminated by an  $r$ -position. A special case is if  $r$  holds at the same position as  $p$  and then the  $q$ -interval is empty. Consequently, the formula states that

every  $p$ -position initiates a sequence of  $q$ -positions terminated by an  $r$ -position, if ever.

- $\square \exists u: (x = u \wedge \bigcirc(x = u + 1))$

The formula refers to a rigid variable  $u$  and a flexible variable  $x$ . It states that at every position  $j$ , there exists a value of  $u$  such that, at position  $j$ ,  $x$  equals  $u$  and, at the next position  $j + 1$ ,  $x$  equals  $u + 1$ . It follows that  $s_{j+1}[x] = s_j[x] + 1$ . This is a way to specify a sequence in which

$x$  increases by 1 from each state to the next.

- $\forall u: \square(x = u \rightarrow \diamond(y = u))$

This formula uses the rigid variable  $u$  to state that every position in which the flexible variable  $x$  equals  $u$  is followed by a position at which the flexible variable  $y$  equals the same value. Thus, this formula states that

every value assumed by  $x$  is eventually assumed by  $y$ .

- $\exists b: (b \wedge \square(b \leftrightarrow \neg \bigcirc b) \wedge \square(p \rightarrow b))$

This formula quantifies over a flexible boolean variable  $b$ . The first two conjuncts state that  $b$  is initially true and its value at each position is the negation of its value in the next position. Consequently,  $b$  is true at all even positions. The last conjunct states that  $p$  can be true only when  $b$  is. Consequently, this formula states that

proposition  $p$  can only be true at even positions.

- $\square(q \rightarrow \Diamond p)$

The formula states that for each position  $j$ , if  $q$  holds at  $j$ , there must be an earlier position  $k \leq j$  satisfying  $p$ . That is,

every  $q$ -position coincides with or is preceded by a  $p$ -position.

## Entailment and Congruence

Two useful derived operators are the *entailment* and *congruence* operators, respectively defined by

$$\begin{array}{lll} p \Rightarrow q & \text{is} & \square(p \rightarrow q), \\ p \Leftrightarrow q & \text{is} & \square(p \leftrightarrow q). \end{array}$$

Note that while the implication  $p \rightarrow q$  holds on  $\sigma$  whenever  $p$  implies  $q$  at the first position of  $\sigma$ , the entailment  $p \Rightarrow q$  requires that  $p$  imply  $q$  at all positions of  $\sigma$ . Thus,  $p \Rightarrow q$  is a stronger type of implication.

In a similar way, the congruence  $p \Leftrightarrow q$  requires that  $p$  equal  $q$  at all positions of  $\sigma$ .

## A Restriction about Quantifiers

For most of our applications, we restrict our attention to formulas in which quantifiers may appear only in the context of state formulas. Thus, we exclude formulas in which a temporal operator appears in the scope of a quantifier. We refer to formulas obeying this restriction as *state-quantified* formulas. From now on, when we refer to a formula, we mean a state-quantified formula. Formulas that do not obey this restriction will be called *temporally-quantified formulas*.

## Nested Waiting-for Formulas

A class of useful properties is described by *nested waiting-for formulas*. These are formulas of the general form

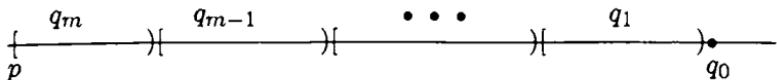
$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0,$$

where  $p, q_0, q_1, \dots, q_m$  are state formulas (assertions). This parentheses-free formula should be interpreted by association to the right, as though it were written

$$p \Rightarrow \left( q_m \mathcal{W} (q_{m-1} \cdots (q_1 \mathcal{W} q_0) \cdots) \right).$$

A model satisfies this formula if every  $p$ -position initiates a succession of intervals, beginning with a  $q_m$ -interval and ending in a  $q_1$ -interval, that may be terminated by a  $q_0$ -position. Each  $q_i$ -interval,  $1 \leq i \leq m$ , is a set of successive positions all of which satisfy  $q_i$ , and may be empty or extend to infinity. In the latter case there need not be a following  $q_j$ -interval, for  $j < i$ , nor a terminating  $q_0$ -position.

This definition can be illustrated by the following figure:



## The Previous and Next Values of an Expression

In some cases, we need to express at position  $j$  the value of an expression as it was at position  $j - 1$ . For a variable  $x$ , we denote the *previous value* of  $x$  by  $x^-$ . For example, we may describe the effect of a transition that increments  $x$  by 1 by the following temporal formula

$$x = x^- + 1.$$

This formula is true at position  $j > 0$  if the value of  $x$  at position  $j$  equals the value of  $x$  at position  $j - 1$  plus 1. Since position 0 has no predecessor, we define the value of  $x^-$  at position 0 to be the same as  $x$ . The formula  $x = x^- + 1$  is therefore false at position 0.

For the special case that  $x$  is a boolean variable, we may compare  $x^-$  to  $\neg x$ . The relation between the two is the following

$$x^- = \neg x \vee (\text{first} \wedge x).$$

The previous-value notation can also be applied to expressions, using the following distribution rule:

$$(f(x_1, \dots, x_k))^- = f(x_1^-, \dots, x_k^-).$$

The previous-value notation is not essential. Whatever can be expressed using this notation can also be expressed without it. In **Problem 0.6**, we ask the reader to show that every formula with occurrences of  $x^-$  can be transformed to an equivalent formula that does not use the previous-value operator.

The previous-value operator is considered to be a past operator. Therefore, it cannot appear in a future or a state formula.

In a symmetric way, we can define the *next value* of a variable or an expression. For a variable  $x$ , we denote the next value of  $x$  by  $x^+$ . Evaluated at position  $j$ ,  $x^+$  yields the value of  $x$  at position  $j + 1$ . Thus, the effect of a transition that increments  $x$  by 1, which was previously described by  $x = x^- + 1$ , can also be described by the temporal formula

$$x^+ = x + 1,$$

which holds at position  $j \geq 0$  if the value of  $x$  at position  $j + 1$  equals the value of  $x$  at position  $j$  plus 1.

In a similar way, we denote by  $e^+$  the value of the expression  $e$  at the next position. The value of  $e^+$  can be evaluated using the following distribution rule:

$$(f(x_1, \dots, x_k))^+ = f(x_1^+, \dots, x_k^+).$$

### Strict Version of the Temporal Operators

The temporal operators as previously defined can be described as *reflexive* in the sense that the present is considered as part of the future (and also of the past). For example,  $\Diamond p$  holding at position  $j$  means that  $p$  holds at some position greater than or equal to  $j$ . In contrast, we may wish to state that  $p$  holds at some position greater than  $j$ . We introduce a set of abbreviations that can be viewed as the *strict versions* of the temporal operators. These are given by:

$$\begin{array}{ll} \widehat{\Box} p = \bigcirc \Box p & \widehat{\ominus} p = \bigcirc \ominus p \\ \widehat{\Diamond} p = \bigcirc \Diamond p & \widehat{\Diamond} p = \ominus \Diamond p \\ p \widehat{\mathcal{U}} q = \bigcirc (p \mathcal{U} q) & p \widehat{\mathcal{S}} q = \ominus (p \mathcal{S} q) \\ p \widehat{\mathcal{W}} q = \bigcirc (p \mathcal{W} q) & p \widehat{\mathcal{B}} q = \ominus (p \mathcal{B} q). \end{array}$$

Note the use of  $\ominus$  in the definitions of  $\widehat{\Diamond}$  and  $\widehat{\mathcal{S}}$ , and the use of  $\bigcirc$  in the definitions of  $\widehat{\Box}$  and  $\widehat{\mathcal{B}}$ .

### Validity, Satisfiability, Equivalence, and Congruence

A state formula (assertion) that holds on all states is called *state valid*, denoted by

$$\Vdash p.$$

For example, associativity of list (or string) concatenation can be stated by the state-valid formula

$$\Vdash x * (y * z) = (x * y) * z.$$

A temporal formula  $p$  is called *satisfiable* if it holds on some model. It is called (*temporally*) *valid*, denoted

$$\models p$$

if it holds on all models. For example, we write

$$\models (\neg p) \mathcal{W} q \vee (\neg q) \mathcal{W} p$$

to state that, either the first  $q$ -state precedes (or coincides with) the first  $p$ -state, or the first  $p$ -state precedes (or coincides with) the first  $q$ -state.

Henceforth, we adopt the convention by which a formula  $p$  that is claimed to be *valid* is state valid if  $p$  is an assertion, and is temporally valid if  $p$  contains at least one temporal operator.

Two formulas  $p$  and  $q$  are defined to be *equivalent*, denoted

$$p \sim q,$$

if the formula  $p \leftrightarrow q$  is valid, i.e.,  $\sigma \models p$  iff  $\sigma \models q$ , for all models  $\sigma$ . For example,

$$p \cup q \sim \Diamond(q \wedge \widehat{\Box}p)$$

claims the equivalence of  $p \cup q$  and  $\Diamond(q \wedge \widehat{\Box}p)$ . Obviously, both formulas claim that there exists a position  $j \geq 0$  such that  $q$  holds at  $j$  and  $p$  holds at all positions  $i$ ,  $0 \leq i < j$ .

Two formulas  $p$  and  $q$  are defined to be *congruent*, denoted

$$p \approx q,$$

if the formula  $p \Leftrightarrow q$  (i.e.,  $\Box(p \leftrightarrow q)$ ) is valid, i.e.,  $(\sigma, j) \models p$  iff  $(\sigma, j) \models q$ , for all models  $\sigma$  and all positions  $j \geq 0$ . If  $p \approx q$  then  $p$  can be replaced by  $q$  in any context, i.e.,  $\varphi(p) \approx \varphi(q)$ , where  $\varphi(p)$  is any formula containing occurrences of  $p$  and  $\varphi(q)$  is obtained from  $\varphi(p)$  by replacing some occurrences of  $p$  by  $q$ . For example,

$$\neg \Diamond p \approx \Box \neg p$$

identifies  $\neg \Diamond p$  and  $\Box \neg p$  as congruent formulas that may be interchanged for one another in any context.

Note that, while  $p \cup q$  and  $\Diamond(q \wedge \widehat{\Box}p)$  are equivalent, they are not congruent. The computation  $\sigma: \langle p: F, q: T \rangle, \langle p: F, q: T \rangle, \dots$  satisfies  $p \cup q$  at position 1 but does not satisfy  $\Diamond(q \wedge \widehat{\Box}p)$  at this position.

The notions of state validity and temporal validity require that a formula hold over all states and all models, respectively. Given a program  $P$ , we can restrict our attention to the set of  $P$ -accessible states and to models that are  $P$ -computations. This leads to several notions of  $P$ -validity.

## *P*-Validity

A state formula  $p$  is called *P-state valid* (state valid over program  $P$ ), denoted

$$P \models p,$$

if it holds over all  $P$ -accessible states (as defined in Section 0.1). For example, we may write

$$P \models x \geq 0$$

to state that variable  $x$  is nonnegative in all states appearing in computations of program  $P$ . Obviously, any state formula that is state valid is also  $P$ -state valid for any program  $P$ .

A temporal formula  $p$  is called *P-valid* (valid over program  $P$ ), denoted

$$P \models p,$$

if it holds over all  $P$ -computations. For example, we may write

$$P \models (x \neq 2) \wedge (x = 1)$$

to state that, in all computations of  $P$ ,  $x$  cannot equal 2 before it equals 1 first. Obviously, any formula that is temporally valid is also  $P$ -valid for any program  $P$ .

In a similar way, we can specialize the notions of satisfiability, equivalence, and congruence to  $P$ -satisfiability,  $P$ -equivalence, and  $P$ -congruence.

For a module  $M$ , we say that formula  $p$  is  *$M$ -valid* (valid over module  $M$ ), denoted

$$M \models p,$$

if  $p$  holds over all computations of  $M$ .

For a top-level process  $P_i$  within program  $P$ , we say that formula  $p$  is *modularly valid over  $P_i$* , denoted

$$P_i \models_m p,$$

if  $p$  holds over all modular computations of  $P_i$ . If  $M_i$  is the module corresponding to  $P_i$  (as defined in Section 0.4), then

$$M_i \models p \text{ iff } P_i \models_m p.$$

## The Four Types of Validity

We summarize the four main notions of validity we have introduced so far:

- For a state formula  $p$ :

$\models p$  —  $p$  is state valid. It holds in all states, e.g.,

$$\models x = 1 \rightarrow x > 0.$$

$P \models p$  —  $p$  is  $P$ -state valid. It holds in all  $P$ -accessible states, e.g.,

$$P \models \text{at\_}\ell_1 \rightarrow x = 0.$$

- For a temporal formula  $p$  (which may also be a state formula):

$\models p$  —  $p$  is valid. It holds in the first state of every model, e.g.,

$$\models \Box p \vee \Diamond \neg p.$$

$P \models p$  —  $p$  is  $P$ -valid. It holds in the first state of every  $P$ -computation, e.g.,

$$P \models \text{at\_}\ell_2 \Rightarrow \Diamond \text{at\_}\ell_4.$$

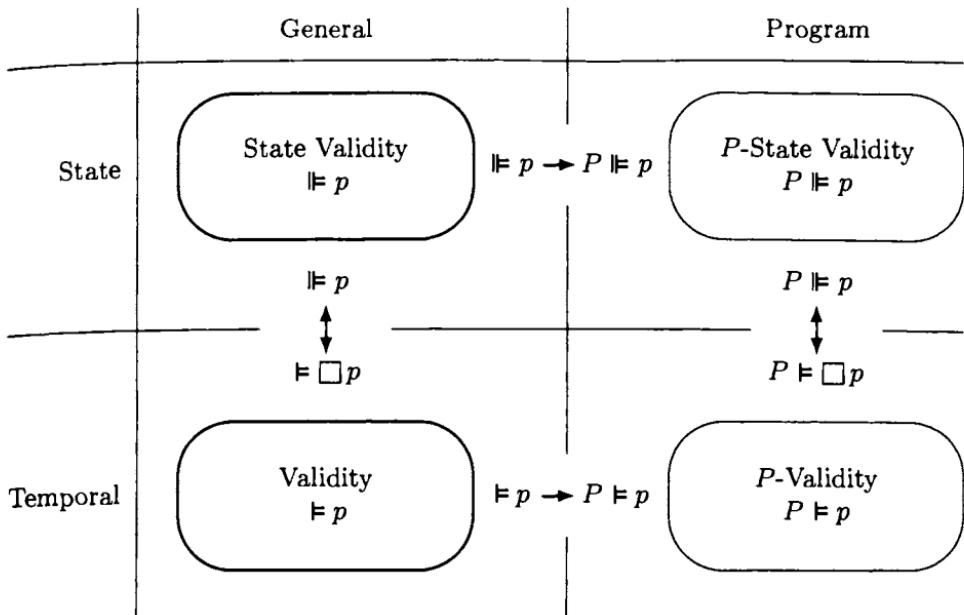


Fig. 0.16. Four types of validity.

In Fig. 0.16, we present a diagram that displays the four notions of validity and the interrelations between them. Two independent parameters together determine the kind of validity. The first parameter distinguishes between state validity and temporal validity. The second parameter distinguishes between validity over general models and validity over  $P$ -computations.

The diagram shows two inclusion relations between general and program validities. These relations can be stated as follows:

- For each state formula  $p$  and program  $P$ , if  $\models p$  then  $P \models p$ ,
- For each (temporal) formula  $p$  and program  $P$ , if  $\models p$  then  $P \models p$ .

Also shown in the diagram are the relations between the state validity of the state formula  $p$  and the temporal validity of the formula  $\Box p$ . These can be summarized as

$$\models p \text{ iff } \models \Box p \quad \text{and} \quad P \models p \text{ iff } P \models \Box p.$$

These relations provide two-way transformations between state and temporal validities.

## Rules and Conventions

Most of our deductions will be based on proof rules. A proof rule has the general form

$$\frac{P_1, \dots, P_k}{C}$$

or, in a more compact representation,

$$P_1, \dots, P_k \vdash C.$$

Lines  $P_1, \dots, P_k$  are the *premises* of the rule, while line  $C$  is the *conclusion*. Each of the lines is a statement of the form  $\triangleright \varphi$  where  $\varphi$  is a temporal formula and  $\triangleright$  is one of the four validity modes  $\Vdash$ ,  $P \Vdash$ ,  $\models$ , or  $P \models$ .

A rule of this form allows us to infer the conclusion from the given premises. As an example, a certain temporal version of modus ponens is given by the following sound rule:

$$\frac{\begin{array}{c} P \Vdash p \\ \Vdash p \rightarrow q \end{array}}{P \models \Box q}$$

This rule states that if assertion  $p$  is  $P$ -state valid and  $p \rightarrow q$  is state valid, then  $\Box q$  is  $P$ -valid.

In many cases, we omit the validity mode  $\triangleright$ . In these cases we adopt the following conventions:

- A proof line containing the assertion  $p$  is to be interpreted as  $P \Vdash p$ , stating the  $P$ -state validity of  $p$ .
- A proof line containing a formula  $\varphi$  which has at least one temporal operator, is to be interpreted as  $P \models p$ , stating the  $P$ -validity of  $\varphi$ .

In **Problem 0.7**, the reader is requested to consider a list of temporal formulas and determine which of them are valid. In **Problem 0.8**, the reader is requested to consider additional temporal operators and show how they can be defined in terms of the existing ones.

## 0.6 Specification of Properties

To specify properties of programs, we extend the temporal language by special control predicates that refer to the location of control within the program. We already introduced the control predicate  $at\_l$ .

We also use the control predicate  $at\_l_{i,j}$  as an abbreviation for the disjunction

$$\text{at\_}\ell_{i,j}: \text{ at\_}\ell_i \vee \text{at\_}\ell_j,$$

and the predicate  $\text{at\_}\ell_{i..j}$  as an abbreviation for

$$\text{at\_}\ell_{i..j}: \text{ at\_}\ell_i \vee \text{at\_}\ell_{i+1} \vee \cdots \vee \text{at\_}\ell_j.$$

For a statement  $\ell: S$ , let  $\ell, \ell_1, \dots, \ell_k$  be the labels of all substatements of  $S$ . We use the control predicate  $\text{in\_}S$  as abbreviation for the disjunction

$$\text{in\_}S: \text{ at\_}\ell \vee \text{at\_}\ell_1 \vee \cdots \vee \text{at\_}\ell_k,$$

denoting that control is somewhere within statement  $S$ .

## Communication Predicates

To specify properties of systems that communicate by message passing, we need the ability to observe that a communication has taken place in the transition leading to the current state. We use two past formulas, which we refer to as *communication predicates*, for this purpose. One of them observes the output of a message on a channel, and the other observes the input of a message from a channel.

We consider separately the cases of asynchronous and synchronous communication.

### Asynchronous Communication

In the case of asynchronous communication, we can detect the events of sending and receiving messages to and from a channel  $\alpha$  by the changes to the corresponding system variable  $\alpha$ . Consequently, we define

$$[\alpha \Leftarrow v]: \neg\text{first} \wedge \alpha = \alpha^- \bullet v,$$

where the expression  $\alpha^- \bullet v$  denotes the list obtained by appending the value  $v$  to the end of the list  $\alpha^-$ .

This predicate, expressed by a past formula, denotes a *sending event*. It holds at position  $j > 0$ , if the transition leading from  $s_{j-1}$  to  $s_j$  sent the value  $v$  to channel  $\alpha$ . We detect this by the fact that  $\alpha$  equals  $\alpha^-$  with  $v$  added to it. This event is caused by the execution of a statement  $\alpha \Leftarrow e$  where the expression  $e$  evaluates to the value  $v$ .

Note that the event  $[\alpha \Leftarrow v]$  reflects communication that occurred only in the very last step leading to the current state. Thus, if  $[\alpha \Leftarrow v]$  is true at position  $j$  of a computation, it will not be true in general at position  $j+1$ , unless the step from  $s_j$  to  $s_{j+1}$  involved another execution of a *send* statement that sent precisely the same value again on channel  $\alpha$ .

In a similar way, we define the *receiving event*

$$[\alpha \triangleright v]: \neg\text{first} \wedge v \bullet \alpha = \alpha^-,$$

where the expression  $v \bullet \alpha$  denotes the list obtained by appending the value  $v$  to the front of the list  $\alpha$ .

This predicate holds at position  $j$  iff  $j > 0$  and the current value of  $\alpha$  equals the previous value  $\alpha^-$  minus its first element, which equals  $v$ . Clearly,  $[\alpha \triangleright v]$  holds precisely at the states in which  $v$  has just been read from channel  $\alpha$ .

This event is caused by taking a transition corresponding to a receive statement  $\alpha \Rightarrow u$ , and the predicate holds only at the state resulting from this transition.

## Synchronous Communication

In the case of synchronous communication, there are no system variables that represent the list of pending messages for each channel. Therefore, to observe that a synchronous communication has taken place, we should look for the activation of the joint transitions that perform synchronous communication.

Let  $\tau_{(\ell,m)}$  be a communication transition associated with the pair of matching statements

$$\ell: \alpha \Leftarrow e; \hat{\ell}: \quad m: \alpha \Rightarrow u; \hat{m}: .$$

Denoting by  $v$  the value of expression  $e$  which this communication assigns to variable  $u$ , we define

$$\text{comm}(\ell, m, v): \quad \{[\ell], [m]\} \subseteq \pi^-$$

$$\wedge \pi = (\pi^- - \{[\ell], [m]\}) \cup \{[\hat{\ell}], [\hat{m}]\} \wedge v = e^-.$$

This formula expresses the fact that the joint transition  $\tau_{(\ell,m)}$  has just been taken and that the value  $v$  of  $e^-$  was communicated.

In a similar way, we can consider the case that statements  $\ell$  and  $m$  are grouped statements containing matching synchronous communication substatements.

$$\ell: \langle T_1; \alpha \Leftarrow e; S_1 \rangle; \hat{\ell}: \quad m: \langle T_2; \alpha \Rightarrow u; S_2 \rangle; \hat{m}: ,$$

where the variables referenced by  $T_1$ ,  $e$ , and  $S_1$  are disjoint from the variables referenced by  $T_2$ ,  $u$ , and  $S_2$ .

For such a pair of matching statements, the  $\text{comm}$  formula is given by

$$\text{comm}(\ell, m, v): \quad \{[\ell], [m]\} \subseteq \pi^-$$

$$\wedge \pi = (\pi^- - \{[\ell], [m]\}) \cup \{[\hat{\ell}], [\hat{m}]\} \wedge v = (e[T_1])^-,$$

where  $e[T_1]$  denotes the value of expression  $e$  in terms of the original state variables, after statement  $T_1$  has been executed. For example, if  $e$  is  $2x$ , and the effect of  $T_1$  is to increase  $x$  by 1, then  $e[T_1]$  equals  $2(x + 1) = 2x + 2$ .

We may now take a disjunction over all matching  $(\ell, m)$  pairs that refer to channel  $\alpha$  and define

$$[\alpha \Leftarrow \Rightarrow v]: \bigvee_{(\ell, m)} \text{comm}(\ell, m, v).$$

Thus, the sending-receiving event  $[\alpha \Leftarrow \Rightarrow v]$  is defined to have taken place iff the last transition taken is a synchronous communication of value  $v$  over channel  $\alpha$ .

Note that synchronous communication consists of a receiving event together with a sending event. Therefore, for synchronous communication, it is sufficient to refer to the single input-output predicate  $[\alpha \Leftarrow \Rightarrow v]$ .

For both synchronous and asynchronous communications, we sometimes need to specify that an input or output on channel  $\alpha$  has taken place, without specifying the value communicated. We define the local predicates

$$[\alpha >] \quad [\alpha <] \quad [\alpha \Leftarrow \Rightarrow ]$$

to hold whenever  $[\alpha > v]$ ,  $[\alpha < v]$ , and  $[\alpha \Leftarrow \Rightarrow v]$  hold for some  $v$ , respectively.

## Specification of Program Properties

A temporal formula  $p$  that is valid over a program  $P$  specifies a property of  $P$ , i.e., states a condition that is satisfied by all  $P$ -computations.

### Example (program PRIME)

We illustrate the expressive power of temporal logic for specifying properties of programs by considering a desired program PRIME, whose role is to print the infinite sequence of all primes in ascending order. We assume that the program prints its output by sending the values to be printed on asynchronous channel  $\alpha$ . Thus, Fig. 0.17 presents the main elements of this program.

```

out α: channel [1..] of integer
loop forever do
  [
    α ← y
    ...
  ]

```

Fig. 0.17. Outline of program PRIME.

The following formulas specify properties that program PRIME should satisfy:

- Only prime numbers are ever printed.  
This property can be expressed by the formula

$$\psi_1: [\alpha \leq u] \Rightarrow \text{prime}(u),$$

where  $u$  is a rigid variable, and  $\text{prime}(u)$  is a predicate that checks if  $u$  is a prime number.

Formula  $\psi_1$  is valid over all computations of program PRIME if and only if its explicitly quantified version, given by

$$\widehat{\psi}_1: \forall u: [\alpha \leq u] \Rightarrow \text{prime}(u),$$

is  $P$ -valid over PRIME. The formula (in both forms) states that if  $u$  is a value sent as output, then  $u$  must be prime.

Note that if this is the only property required, then a program that outputs the integer sequence

$$2, 11, 5, 13, \dots,$$

would be acceptable.

- The output sequence should be monotonically increasing.  
This property can be expressed by the formula

$$\psi_2: [\alpha \leq u] \wedge \widehat{\Diamond}[\alpha \leq v] \Rightarrow u > v,$$

where  $u$  and  $v$  are rigid variables.

Requirements  $\psi_1$  and  $\psi_2$  together still allow the following output sequence, which should not be acceptable since it skips some primes.

$$2, 11, 17, 29, \dots.$$

- Every prime should eventually be printed.  
This property can be expressed by the formula

$$\psi_3: \text{prime}(u) \rightarrow \Diamond[\alpha \leq u],$$

where  $u$  is a rigid variable.

Only  $\psi_1$ ,  $\psi_2$ , and  $\psi_3$  together provide a satisfactory specification of program PRIME. ■

## Classification of Formulas

The temporal formulas that specify program properties can be arranged in a hierarchy that identifies several classes of formulas differing in their expressive power.

For past formulas  $p$ ,  $p_i$ , and  $q_i$ , we define the following classes of canonical formulas:

$\square p$	is a canonical <i>safety</i> formula
$\diamondsuit p$	is a canonical <i>guarantee</i> formula
$\bigwedge_{i=1}^n [\square p_i \vee \diamondsuit q_i]$	is a canonical <i>obligation</i> formula
$\square \diamondsuit p$	is a canonical <i>response</i> formula
$\diamondsuit \square p$	is a canonical <i>persistence</i> formula
$\bigwedge_{i=1}^n [\square \diamondsuit p_i \vee \diamondsuit \square q_i]$	is a canonical <i>reactivity</i> formula.

In the following, let  $\kappa$  range over the class names:

*safety, guarantee, obligation, response, persistence, reactivity.*

A formula is called a  $\kappa$ -formula if it is equivalent to a canonical  $\kappa$ -formula.

**Example** For state formulas  $p$  and  $q$ ,

- $p \Rightarrow \square q$  and  $p \mathcal{W} q$  are safety formulas, due to the equivalences

$$\begin{aligned} p \Rightarrow \square q &\sim \square(\diamondsuit p \rightarrow q) \\ p \mathcal{W} q &\sim \square(\diamondsuit \neg p \rightarrow \diamondsuit q). \end{aligned}$$

- $p \rightarrow \diamondsuit q$  and  $p \mathcal{U} q$  are guarantee formulas, due to the equivalences

$$\begin{aligned} p \rightarrow \diamondsuit q &\sim \diamondsuit(\diamondsuit(first \wedge p) \rightarrow q) \\ p \mathcal{U} q &\sim \diamondsuit(q \wedge \hat{\square} p). \end{aligned}$$

- $p \mathcal{W} (\diamondsuit q)$  is an obligation formula, due to the equivalence

$$p \mathcal{W} (\diamondsuit q) \sim \square p \vee \diamondsuit q.$$

- $p \Rightarrow \diamondsuit q$  is a response formula, due to the equivalence

$$p \Rightarrow \diamondsuit q \sim \square \diamondsuit((\neg p) \mathcal{B} q).$$

- $p \Rightarrow \diamondsuit \square q$  is a persistence formula, due to the equivalence

$$p \Rightarrow \diamondsuit \square q \sim \diamondsuit \square(\diamondsuit p \rightarrow q).$$

- $\square \diamondsuit p \Rightarrow \diamondsuit q$  is a reactivity formula, due to the equivalence

$$\square \diamondsuit p \Rightarrow \diamondsuit q \sim \square \diamondsuit q \vee \diamondsuit \square \neg p. \blacksquare$$

For example, formulas  $\psi_1$  and  $\psi_2$ , which were used to specify properties of program PRIME, are safety formulas since they can be written as

$$\underbrace{[\alpha \lessdot u] \Rightarrow prime(u)}_{\psi_1} \sim \square([\alpha \lessdot u] \rightarrow prime(u))$$

$$\underbrace{[\alpha \leq u] \wedge \widehat{\Diamond} [\alpha \leq v]}_{\psi_2} \Rightarrow u > v \sim \Box ([\alpha \leq u] \wedge \widehat{\Diamond} [\alpha \leq v] \rightarrow u > v).$$

Similarly,  $\psi_3$  is a guarantee formula since it is of the form

$$\psi_3: \underbrace{\text{prime}(u)}_p \rightarrow \Diamond \underbrace{[\alpha \leq u]}_q.$$

## Classification Diagram

The hierarchy of temporal formulas can be displayed in a diagram as shown in Fig. 0.18.

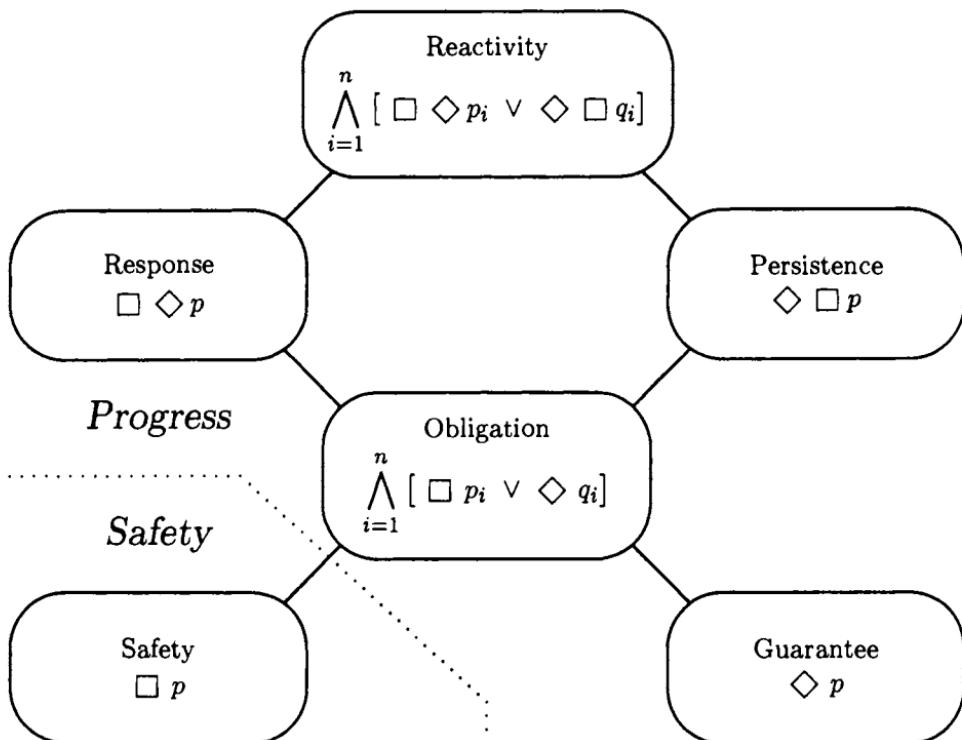


Fig. 0.18. Classification diagram.

Nodes in the diagram represent the different classes of formulas, where each node is labeled by the canonical formula of the class. Edges represent strict inclusion holding between the classes. Thus, the class of safety formulas is strictly included in the class of obligation formulas.

It can be shown that every (state-quantified) temporal formula is equivalent to a canonical reactivity formula. Therefore, the diagram of Fig. 0.18 actually classifies all the temporal formulas. As shown in the diagram, we refer to the non-safety classes as the *progress* classes.

In this volume we only consider safety formulas. The next volume, the PROGRESS book, presents rules and methods for verifying properties specified by progress formulas.

## Classification of Properties

We define a *property*  $\Pi$  to be a set of models. A property is said to be *specified* by a temporal formula  $p$  if, for every model  $\sigma$ ,  $\sigma \in \Pi$  iff  $\sigma \models p$ . A property is called a  $\kappa$ -*property* if it is specifiable by a  $\kappa$ -formula.

A program  $P$  is said to *have property*  $\Pi$  if the set of  $P$ -computations is a subset of  $\Pi$ . If  $\Pi$  is specified by the formula  $p$ , then this corresponds to the case that  $p$  is  $P$ -valid.

The hierarchy of classes of formulas induces an identical hierarchy of properties. Thus, any safety property is also an obligation, response, persistence and reactivity property.

In **Problem 0.9**, we request the reader to show that each class of formulas is closed under conjunction and disjunction. In **Problem 0.10**, we request the reader to consider an extension of the canonical formulas.

## Properties of Terminating Programs

Consider a program  $P$  which is expected to terminate and produce some final result on termination. With such a program, we can often associate a *postcondition*  $\psi$  describing the desired relation between the final results and the inputs to the program. Frequently, we also specify a *precondition*  $\varphi$  which expresses constraints on the inputs, for which the program is expected to behave properly.

Consider, for example, a program SQRT that should compute in variable  $y$  the square root of a real number, initially placed in variable  $x$ , up to accuracy  $\epsilon > 0$ . The precondition and postcondition that may be associated with this program are

$$\varphi: x \geq 0 \quad \text{and} \quad \psi: |y - \sqrt{x}| \leq \epsilon.$$

The precondition  $\varphi$  identifies the nonnegative reals as acceptable inputs to the program. The postcondition  $\psi$  specifies that the output of the program should be  $\sqrt{x}$  within accuracy  $\epsilon$ .

Assume that the program has the following standard form:

$$P ::= \left[ \text{declaration}; [P_1 :: [\ell_1: S_1; \widehat{\ell}_1: ]] \parallel \cdots \parallel P_k :: [\ell_k: S_k; \widehat{\ell}_k: ] \right]$$

We denote by  $\widehat{L}$  the set of locations  $\{\widehat{\ell}_1, \dots, \widehat{\ell}_k\}$ , and introduce the abbreviation:

$$\text{at-}\widehat{L}: \text{at-}\widehat{\ell}_1 \wedge \cdots \wedge \text{at-}\widehat{\ell}_k,$$

characterizing states in which all top-level processes are at their terminal locations.

A state  $s$  is defined to be *terminal* for a program  $P$  if it satisfies  $\text{at-}\widehat{L}$ . Obviously, the only transition enabled on a terminal state is the idling transition  $\tau_I$ . A computation that contains a terminal state  $s$  is said to *terminate in  $s$*  and is called a *terminating computation*. A computation that does not terminate is called a *divergent computation*.

Of particular interest are computations whose initial state satisfies the precondition  $\varphi$ . We refer to such computations as  *$\varphi$ -computations*.

There are three standard properties (requirements) that can be posted for a program that is expected to terminate:

### Partial Correctness

The property of *partial correctness* requires that every terminating  $\varphi$ -computation terminate in a  $\psi$ -state. Thus, no guarantee of termination is given, but if a  $\varphi$ -computation terminates, it is required to terminate in a  $\psi$ -state (a state satisfying the postcondition  $\psi$ ).

Partial correctness can be specified by the following formula:

$$\varphi \rightarrow \square(\text{at-}\widehat{L} \rightarrow \psi),$$

which states that, if  $\varphi$  holds initially, then whenever the program reaches the terminal location set  $\widehat{L}$ ,  $\psi$  holds.

The property of partial correctness is a safety property because it can be specified by the following canonical safety formula:

$$\square(\text{at-}\widehat{L} \wedge \Diamond(\text{first} \wedge \varphi) \rightarrow \psi).$$

### Total Correctness

The property of *total correctness* requires that every  $\varphi$ -computation terminate in a  $\psi$ -state. Thus, total correctness guarantees termination, as well as termination in a good ( $\psi$ )-state. This property can be specified by the following formula:

$$\varphi \rightarrow \Diamond(\text{at-}\widehat{L} \wedge \psi),$$

which claims that every initial  $\varphi$ -state is followed by a terminating  $\psi$ -state.

As shown above, a formula of this form specifies a guarantee property.

## Termination

The property of *termination* requires that every  $\varphi$ -computation terminate. It can be specified by the formula:

$$\varphi \rightarrow \Diamond \text{at\_}\widehat{L},$$

which claims that every initial  $\varphi$ -state is followed by a terminal state.

As shown above, a formula of this form specifies a guarantee property.

## 0.7 Overview of the Verification Framework

Our framework for verifying temporal properties of reactive systems is based on the following four components:

1. A *computational model* to describe the behavior of reactive systems.
2. A *system description language* to express proposed implementations.
3. A *specification language* to express properties that any proposed implementation should satisfy.
4. *Verification techniques* to prove that the proposed implementation meets its intended specification.

As a computational model, we take Fair Transition Systems (FTS). This semantic model can be conveniently connected to many system description languages, with widely divergent syntactic presentations.

As a system description language, we use our Simple Programming Language (SPL). While SPL does not have all the conveniences of popular programming languages, it does offer a simple testbed to illustrate the verification methodology that is advocated here. Since all of the rules are based on the fair transition system model, the particular choice of system description language has no strong impact on the verification methodology. Indeed, one of the reasons for introducing the fair transition system is to abstract away the details of programming languages and offer a uniform representation for all reactive systems.

As a specification language, we use Temporal Logic (TL), a suitable language for specifying reactive systems, which abstracts away timing measures. As explained earlier, computations of a fair transition system, being sequences of states, are a subset of the models over which temporal formulas are interpreted. This leads to a natural way of specifying properties of reactive systems by temporal logic.

Obviously, formal correctness of a system only guarantees that it satisfies the specification. It is therefore important that the specification itself captures accurately the informal intentions and requirements of the specifiers. If the spec-

ification language is cumbersome and difficult to read, then there is a greater chance that the requirements specification is incorrect. The advantage of temporal logic over alternative formalisms, such as first-order logic, is that many useful properties can be expressed in a shorter and more readable form. For example, the property “for every occurrence of  $p$  there is a subsequent occurrence of  $q$  and an intermediary occurrence of  $r$  between the two” can be expressed in temporal logic as

$$\square(p \rightarrow \diamond[r \wedge \diamond q]).$$

The same property expressed in first-order logic would be

$$\forall t_1 \geq 0: [p(t_1) \rightarrow \exists t_2: (t_1 \leq t_2 \wedge r(t_2) \wedge \exists t_3: [t_2 \leq t_3 \wedge q(t_3)])],$$

where the  $t$ 's are timing variables.

Figure 0.19 illustrates how the four parts of the verification framework fit together. We have already presented the first three parts of the framework, namely, the FTS computational model, the SPL system description language, and the TL specification language. The rest of the book presents a set of verification techniques for establishing that a reactive system (program) satisfies its temporal specification, discusses methods for applying them, and illustrates their use on representative examples.

Although temporal logic offers a concise style of specification, there is a cost. Reasoning in temporal logic is more complex and certainly less familiar than first-order reasoning. While pure temporal-logic reasoning is harder than first-order reasoning, we alleviate this difficulty by providing rules that derive temporal conclusions from first-order properties. That is, we reduce the problem of verifying a temporal property over a program to verifying a finite set of first-order properties over the same program. (Even in the few cases where our rules have temporal premises, we either show how to “statify” the premises so that reasoning is no harder than first-order reasoning, or derive the premises using other simpler verification rules, which ultimately have only first-order premises.) Our reliance on first-order logic is illustrated in the relative completeness results, which claim that any temporal formula that holds over all computations of a program  $P$  can be proven to be  $P$ -valid, using the verification rules presented in this book and an oracle for verifying first-order formulas.

Using the classification of temporal properties, the proposed proof system is organized by property classes. Each rule addresses properties belonging to a particular class.

In Chapters 1–4 of this volume, we present proof rules and methods for the verification of safety properties. According to the classification of properties, a *safety property* is a property that can be specified by a formula of the form  $\square p$  for some past formula  $p$ . In Chapters 1 and 2, we consider the special case of properties specifiable by a formula  $\square p$ , where  $p$  is a state formula (assertion).

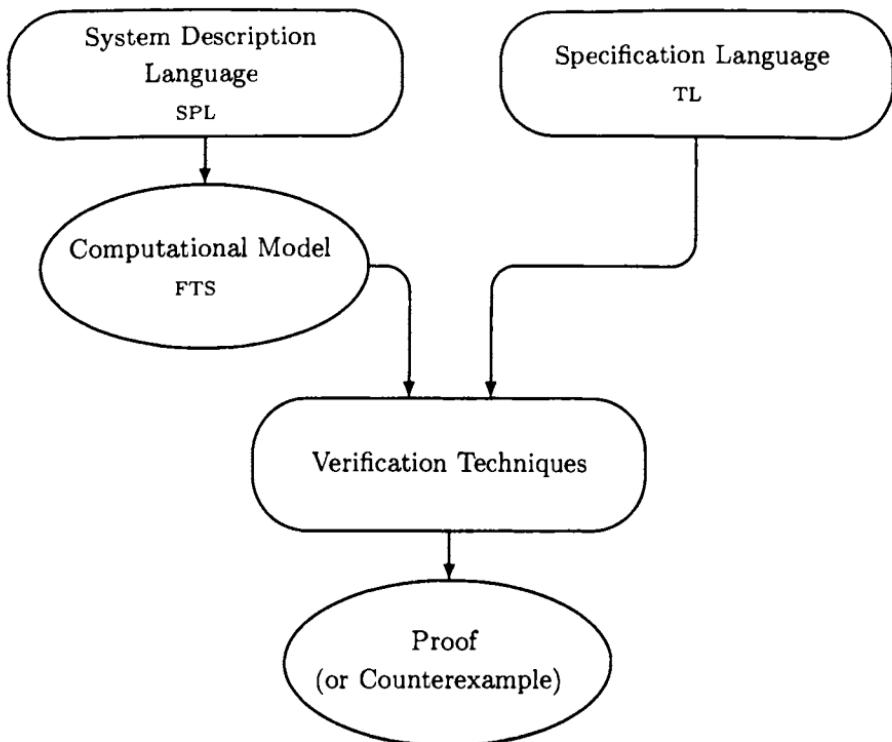


Fig. 0.19. Overview of the verification process.

Chapter 1 presents the basic methods for proving such properties, while Chapter 2 illustrates their application to more advanced examples and develops related theories. Chapter 3 considers properties that are specifiable by a nested waiting-for formula. Chapter 4 concludes the treatment of safety properties by presenting rules for verifying properties expressible by a formula  $\square p$ , where  $p$  is an arbitrary past formula.

The PROGRESS book will complete the presentation of the verification framework by presenting rules and algorithms for all the progress properties.

We should point out that the difficulty of verification lies in finding appropriate intermediate formulas and other constructs that validate the premises of some rule and yield the desired conclusion. Such a process is analogous to the process of finding a strong enough inductive hypothesis in a mathematical proof by induction. While such a search for inductive assertions is undecidable in general, we present several heuristics intended to make the task easier. These heuristics assume that the verifier has a good understanding of what the system does, since the inductive intermediary assertions often represent additional facts about

the behavior of the system which can be provided only by somebody with such understanding.

The deductive approach is based on verification rules that require ingenuity in finding auxiliary constructs. In addition, we also consider *algorithmic verification*, which is based on algorithms that, without any user guidance or ingenuity, check automatically whether a given program (system) satisfies its temporal specification. The obvious advantages of this approach are offset by the fact that it can only be applied to *finite-state systems*, i.e., systems restricted to having only finitely many distinct states.

Our study of the algorithmic approach is also organized by property classes. At the end of Chapter 2, we present an algorithm for checking whether a finite-state system satisfies a property specifiable by a formula  $\Box p$  for an assertion  $p$ . The last section in Chapter 3 presents an algorithm for automatic verification of nested waiting-for formulas. The last section of Chapter 4 presents an algorithm for automatic verification of formulas  $\Box p$  for a past formula  $p$ .

All these individual algorithms apply only to canonical formulas. The final Chapter of this volume, Chapter 5, presents a universal algorithm that checks whether a finite-state system satisfies a general (state-quantified) temporal formula.

## Problems

**Problem 0.1** (an irreproducible while statement) page 24

Consider program SB of Fig. 0.20.

(a) Identify the locations of this program as equivalence classes of labels. List the post-location of each of the statements.

(b) Show that this program has a terminating computation.

This terminating computation may appear to be counterintuitive. One way out of this difficulty is to disallow while statements as children of selection statements.

Another solution is to consider a different version of a *while* statement and require that the body of a *while* statement always have a post-label that is not equivalent to other labels. We may present this version as

$$\ell_1: [\text{WHILE } c \text{ DO } [\ell_2: S; \widehat{\ell}_2:]]; \ell_3:,$$

where  $\widehat{\ell}_2 \not\sim_L \ell_1$ .

(c) Define transitions and transition relations for this version of a WHILE statement. Show that the version of program SB in which the *while* statement has been replaced by this WHILE statement has no terminating computation.

**out**  $x$ : integer **where**  $x = 0$

$\ell_0:$   $\left[ \begin{array}{l} [\ell_1: \text{while } x \geq 0 \text{ do } \ell_2: x := x + 1] \\ \text{or} \\ [\ell_3: \text{await } x > 0] \end{array} \right]$

$\ell_4:$

Fig. 0.20. Program SB (strange behavior).

**Problem 0.2** (computing any natural number) page 29

- (a) Consider program ANY-NAT presented in Fig. 0.21. Argue (informally) that this program always terminates. Also show that, for every natural number  $n \geq 0$ , there exists a computation of ANY-NAT such that  $y = n$  on termination.

**out**  $y$ : integer **where**  $y = 0$   
**local**  $x$ : boolean **where**  $x = \text{T}$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{while } x \text{ do} \\ \quad \ell_1: y := y + 1 \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: x := \text{F} \\ m_1: \end{array} \right]$

Fig. 0.21. Program ANY-NAT (computing any natural number).

- (b) Construct a program with a single process that exhibits a similar behavior; that is, all of its computations terminate and, for each natural number  $n$ , there exists a computation producing  $n$ .

*Hint:* Consider the effect of a compassionate statement, such as a *receive*, at the head of a selection statement.

- \* (c) Prove that no single-process program with only just transitions can have the same behavior as ANY-NAT.

\* **Problem 0.3** (congruent statements) page 29

Consider a program  $P$  of the form

$$P ::= \left[ \text{declaration}; [[\ell_1: S_1; \widehat{\ell}_1: ]] \parallel \cdots \parallel [\ell_k: S_k; \widehat{\ell}_k: ]] \right]$$

A state  $s_j$  in a computation of  $P$  is said to be *terminal* if it satisfies  $\text{at\_}\widehat{\ell}_1 \wedge \cdots \wedge \text{at\_}\widehat{\ell}_k$ . Obviously, if  $s_j$  is terminal, then so is  $s_i$  for every  $i \geq j$ .

Let  $P_1$  and  $P_2$  be two programs whose system variables  $V_1$  and  $V_2$  may differ at most by their local variables, i.e., they have equal *in* and *out* variables. Let  $V = V_1 \cap V_2$  denote the set of variables that are common to  $P_1$  and  $P_2$ . We say that  $P_1$  and  $P_2$  *agree on their initial states* if, for every  $s_1$ , an initial state of  $P_1$ , there exists  $s_2$ , an initial state of  $P_2$ , such that  $s_1$  and  $s_2$  agree on the interpretation of  $V$ . Similarly, for every  $s_2$ , an initial state of  $P_2$ , there exists  $s_1$  an initial state of  $P_1$  which agrees with  $s_2$  on  $V$ . A state is called a *common initial state* if it is an initial state of both  $P_1$  and  $P_2$ .

Programs  $P_1$  and  $P_2$  are called *termination equivalent* if they agree on their initial states and, for each common initial state  $s$ ,

- $P_1$  has a divergent  $s$ -computation iff  $P_2$  has such a computation.
- $P_1$  has an  $s$ -computation terminating in state  $\widehat{s}_1$  iff  $P_2$  has an  $s$ -computation terminating in state  $\widehat{s}_2$  such that  $\widehat{s}_1$  and  $\widehat{s}_2$  agree on  $V$ .

Let  $P[S]$  be a program containing a single occurrence of the statement symbol  $S$ . We refer to this parameterized program as a *context*. Consider, for example the following context,

$$P[S] ::= [\text{out } x: \text{integer where } x = 0; S]$$

The statements  $S_1$  and  $S_2$  are defined to be *congruent* if the two programs  $P[S_1]$  and  $P[S_2]$ , obtained by replacing the symbol  $S$  by  $S_1$  and  $S_2$  respectively, are termination equivalent for every context  $P[S]$ .

(a) Consider the two statements

$$S_1 ::= [x := 1; x := 2]$$

$$S_2 ::= [x := 1; x := x + 1].$$

and the context

$$P[S] ::= [\text{out } x: \text{integer where } x = 0; S].$$

Show that  $P[S_1]$  and  $P[S_2]$  are termination equivalent.

(b) Consider the preceding two statements and the context

$$Q ::= [\text{out } x: \text{integer where } x = 0; [S \parallel x := 0]].$$

Show that  $Q[S_1]$  and  $Q[S_2]$  are not termination equivalent. We may conclude that  $S_1$  and  $S_2$  are not congruent.

(c) Show the following congruences

1.  $[S_1 \text{ or } S_2] \approx [S_2 \text{ or } S_1]$

2.  $[S_1 \parallel S_2] \approx [S_2 \parallel S_1]$
3.  $S \approx [S; \text{skip}]$
4.  $\text{await } c \approx \text{while } \neg c \text{ do skip}$

(d) Are the two statements

$\text{await } x$  and  $\text{skip}; m: \text{await } x$

congruent? Prove your answer.

(e) Let  $y$  be a boolean variable. Which of the three following statements are congruent?

$y := T$

$\text{if } y \text{ then } y := T \text{ else } y := T$

$\left[ [\text{await } y; y := T] \text{ or } [\text{await } \neg y; y := T] \right]$

#### Problem 0.4 (mutual exclusion) page 32

Program TRY-MUX1 of Fig. 0.22 is suggested as a tentative solution to the mutual-exclusion problem. As explained in pages 29–33, a good solution to the mutual-exclusion problem is expected to satisfy the requirements of exclusion and accessibility.

(a) Is program TRY-MUX1 a good solution to the mutual-exclusion problem? If your answer is positive, argue (informally) that the program satisfies the requirements of exclusion and accessibility. If your answer is negative, indicate which requirement is not satisfied by showing a computation that violates it. If the violated requirement is that of accessibility, does the program satisfy the weaker property of communal accessibility?

local  $y_1, y_2$ : integer where  $y_1 = 0, y_2 = 0$

$P_1 ::$ <div style="border: 1px solid black; padding: 10px; display: inline-block;"> <math>\ell_0: \text{loop forever do} \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{await } y_2 = 0 \\ \ell_3: y_1 := 1 \\ \ell_4: \text{critical} \\ \ell_5: y_1 := 0 \end{array} \right]</math> </div>	$P_2 ::$ <div style="border: 1px solid black; padding: 10px; display: inline-block;"> <math>\ell_0: \text{loop forever do} \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{await } y_1 = 0 \\ \ell_3: y_2 := 1 \\ \ell_4: \text{critical} \\ \ell_5: y_2 := 0 \end{array} \right]</math> </div>
---	---

Fig. 0.22. Program TRY-MUX1 (proposed solution to mutual exclusion).

- (b) The same questions for TRY-MUX2, a version of program TRY-MUX1 in which statements  $\ell_2$  and  $\ell_3$  are interchanged and so are statements  $m_2$  and  $m_3$ .
- (c) The same questions for program TURN of Fig. 0.23.

**Problem 0.5** (message-passing mutual exclusion) page 36

Consider program MUX-SYNCH, presented in Fig. 0.24. This program attempts to solve the mutual-exclusion problem in an architecture in which processes communicate by synchronous message passing. The program consists of two processes  $P_1$  and  $P_2$ , which compete on entering their critical sections, and a process  $A$  which arbitrates between the two.

local  $t$ : integer where  $t = 1$

$$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{await } t = 1 \\ \ell_3: \text{critical} \\ \ell_4: t := 2 \end{array} \right] \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \text{await } t = 2 \\ m_3: \text{critical} \\ m_4: t := 1 \end{array} \right] \end{array} \right]$$

Fig. 0.23. Program TURN (turn taking).

- (a) Argue (informally) that program MUX-SYNCH is a good solution to the mutual-exclusion problem. That is, show that each computation of the program satisfies the requirements of exclusion and accessibility.
- (b) Consider a non-standard transition system for program MUX-SYNCH, in which transitions associated with communication statements are taken to be just but not compassionate. We refer to this interpretation of the program as MUX-SYNCH-J. Is this program (transition system) a good solution to the mutual-exclusion problem? Provide an informal argument in support of a positive answer or, alternately, show a computation that violates one of the two requirements.
- (c) Consider program MUX-ASYNCH which is obtained from program MUX-SYNCH by redeclaring channels  $\alpha_1$ ,  $\alpha_2$ ,  $\beta_1$ , and  $\beta_2$  as asynchronous channels. Is program MUX-ASYNCH a good solution to the mutual-exclusion problem? Provide an informal argument in support of a positive answer or, alternately, show a computation that violates one of the two requirements.
- (d) In Fig. 0.25, we present program MUX-SHARED which attempts to simulate program MUX-SYNCH by shared-variables communication. Variables  $\alpha_1$ ,  $\alpha_2$ ,  $\beta_1$ ,

**local**  $\alpha_1, \alpha_2, \beta_1, \beta_2$ : channel of boolean

$P_1 :: \left[ \begin{array}{l} \text{local } x_1: \text{boolean} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \alpha_1 \Leftarrow T \\ \ell_3: \beta_1 \Rightarrow x_1 \\ \ell_4: \text{critical} \\ \ell_5: \alpha_1 \Leftarrow F \end{array} \right] \end{array} \right]$

||

$A :: \left[ \begin{array}{l} \text{local } y: \text{boolean} \\ k_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} [k_1^a: \alpha_1 \Rightarrow y; k_2: \beta_1 \Leftarrow T; k_3: \alpha_1 \Rightarrow y] \\ \text{or} \\ [k_1^b: \alpha_2 \Rightarrow y; k_4: \beta_2 \Leftarrow T; k_5: \alpha_2 \Rightarrow y] \end{array} \right] \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} \text{local } x_2: \text{boolean} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \alpha_2 \Leftarrow T \\ m_3: \beta_2 \Rightarrow x_2 \\ m_4: \text{critical} \\ m_5: \alpha_2 \Leftarrow F \end{array} \right] \end{array} \right]$

Fig. 0.24. Program MUX-SYNCH (mutual exclusion by synchronous communication).

and  $\beta_2$  are redeclared as boolean variables, sending is replaced by setting a variable, and receiving is replaced by waiting for a variable to assume an expected value.

Is program MUX-SHARED a good solution to the mutual-exclusion problem? Present an informal argument or show a computation that violates one of the two requirements.

**local**  $\alpha_1, \alpha_2, \beta_1, \beta_2$ : boolean **where**  $\alpha_1 = \alpha_2 = \beta_1 = \beta_2 = F$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \alpha_1 := T \\ \ell_3: \text{await } \beta_1 \\ \ell_4: \text{critical} \\ \ell_5: \alpha_1 := F \\ \ell_6: \text{await } \neg\beta_1 \end{array} \right] \end{array} \right]$

||

$A :: \left[ \begin{array}{l} k_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} [k_1^a: \text{await } \alpha_1; k_2: \beta_1 := T; k_3: \text{await } \neg\alpha_1; k_4: \beta_1 := F] \\ \quad \text{or} \\ [k_1^b: \text{await } \alpha_2; k_5: \beta_2 := T; k_6: \text{await } \neg\alpha_2; k_7: \beta_2 := F] \end{array} \right] \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \alpha_2 := T \\ m_3: \text{await } \beta_2 \\ m_4: \text{critical} \\ m_5: \alpha_2 := F \\ m_6: \text{await } \neg\beta_2 \end{array} \right] \end{array} \right]$

Fig. 0.25. Program MUX-SHARED (mutual exclusion by shared variables).

**Problem 0.6** (previous-value operator not essential) page 49

Let  $\varphi$  be a formula containing one or more occurrences of expressions of the form  $x^-$ . Show that there exists a formula  $\psi$ , equivalent to  $\varphi$ , that does not use the previous-value operator but may use quantification over rigid variables.

**Problem 0.7** (valid and invalid formulas) page 54

The following list of temporal congruences and equivalences contains valid and invalid formulas. Break each congruence into two entailments and each equiva-

lence into two implications. For each entailment or implication, decide if it is valid or not. For entailments and implications claimed to be valid, give an informal (semantic) justification. For an entailment or implication  $\varphi$  claimed to be invalid, describe a sequence  $\sigma$ , such that  $\sigma \not\models \varphi$ .

- (a)  $\Diamond p \wedge \Box q \Leftrightarrow \Diamond(p \wedge \Box q)$
- (b)  $\Diamond p \wedge \Box q \Leftrightarrow \Box(\Diamond p \wedge q)$
- (c)  $\Diamond \Box p \wedge \Diamond \Box q \Leftrightarrow \Diamond(\Box p \wedge \Box q)$
- (d)  $(p \mathcal{U} q) \mathcal{U} q \Leftrightarrow p \mathcal{U} q$
- (e)  $p \mathcal{U} q \Leftrightarrow ((\neg p) \mathcal{U} q \rightarrow p \mathcal{U} q)$
- (f)  $p \mathcal{U} q \wedge q \mathcal{U} r \Leftrightarrow p \mathcal{U} r$
- (g)  $\Diamond p \Leftrightarrow \Box(\Diamond p \vee \Diamond \Diamond p)$
- (h)  $\Diamond \Diamond p \Leftrightarrow \Diamond \Diamond p$
- (i)  $(q \Rightarrow \Diamond p) \Leftrightarrow (\neg q) \mathcal{W} p$
- (j)  $(\Box p \vee \Box q) \Leftrightarrow \Box(\Box p \vee \Box q)$
- (k)  $(p \rightarrow \Box q) \Leftrightarrow \Box(\Diamond p \rightarrow q)$
- (l)  $\Diamond(p \wedge \Diamond q) \Leftrightarrow \Diamond(q \wedge \Diamond p)$
- (m)  $(\Diamond p \wedge \Diamond q) \Leftrightarrow \Diamond(\Diamond p \wedge \Diamond q)$
- (n)  $(\Box p \vee \Diamond q) \Leftrightarrow p \mathcal{W} (\Diamond q)$
- (o)  $(p \Rightarrow \Diamond q) \Leftrightarrow \Box \Diamond((\neg p) \mathcal{B} q)$
- (p)  $(p \Rightarrow \Diamond q) \Leftrightarrow \Box \Diamond((\neg p) \mathcal{B} q)$
- (q)  $(\Box \Diamond p \wedge \Box \Diamond q) \Leftrightarrow \Box \Diamond(q \wedge (\neg q) \widehat{\mathcal{B}} p)$
- (r)  $\Diamond p \Leftrightarrow \Diamond \Diamond p$
- (s)  $(p \Rightarrow \Diamond \Box q) \Leftrightarrow \Diamond \Box(\Diamond p \rightarrow q)$
- (t)  $(\Diamond \Box p \vee \Diamond \Box q) \Leftrightarrow \Diamond \Box(q \vee p \widehat{\mathcal{B}}(p \wedge \neg q))$
- (u)  $\Diamond \Box(p \rightarrow \Box q) \Leftrightarrow (\Diamond \Box q \vee \Diamond \Box \neg p)$
- (v)  $\bigcirc \bigcirc p \Leftrightarrow \Box((\Theta \Theta \text{first}) \rightarrow p)$
- (w)  $p \mathcal{U} q \Leftrightarrow \Diamond(q \wedge \Box p)$
- (x)  $(p \Rightarrow (\neg q) \mathcal{W} r) \Leftrightarrow (q \Rightarrow (\neg p) \mathcal{B} r)$
- (y)  $\neg(p \mathcal{U} q) \Leftrightarrow (\neg q) \mathcal{W} (\neg p \wedge \neg q)$
- (z)  $\neg(p \mathcal{U} q) \Leftrightarrow (\neg q) \mathcal{W} (\neg p \wedge \neg q).$

**Problem 0.8** (derived temporal operators) page 54

Derived temporal operators can sometimes express properties in a more concise form than the standard operators. We introduce two such operators:

- The *precedence* operator  $\mathcal{P}$  can be defined as

$$p \mathcal{P} q = (\neg q) \mathcal{W} (p \wedge \neg q).$$

- (a) Give a semantic definition for  $(\sigma, j) \models p \mathcal{P} q$  in the style given for  $\mathcal{U}$  in page 44.
- (b) Show how to express  $\mathcal{U}$  in terms of  $\mathcal{P}$  and the boolean connectives.

- The *while* operator  $\mathcal{W}$  can be defined as

$$(\sigma, j) \models p \mathcal{W} q \text{ iff for every } k \geq j, \text{ if } (\sigma, i) \models q \text{ for all } i, j \leq i \leq k, \text{ then } (\sigma, k) \models p.$$

- (c) Can  $\mathcal{W}$  be expressed in terms of  $\mathcal{U}$  and the boolean connectives?
- (d) Can  $\mathcal{U}$  be expressed in terms of  $\mathcal{W}$  and the boolean connectives?

**Problem 0.9** (closure of the property classes) page 61

Let  $\kappa$  range over the following names:

safety, guarantee, obligation, response, persistence, reactivity.

Show that the class of  $\kappa$ -formulas is closed under conjunction and disjunction. That is, show that if  $p_1$  and  $p_2$  are  $\kappa$ -formulas then so are  $p_1 \wedge p_2$  and  $p_1 \vee p_2$ .

**Problem 0.10** (extended canonical formula) page 61

A formula is called an *extended past formula* if the only future temporal operator it may contain is  $\bigcirc$ .

Let  $\kappa$  range over the following names:

safety, guarantee, obligation, response, persistence, reactivity.

We define *extended  $\kappa$ -canonical formulas* analogously to the canonical  $\kappa$ -formulas defined on page 58, where  $p$ ,  $p_i$ , and  $q_i$  are now extended past formulas.

Show that an extended canonical  $\kappa$ -formula specifies a  $\kappa$ -property. That is, every extended canonical  $\kappa$ -formula is equivalent to some canonical  $\kappa$ -formula.

## Bibliographic Remarks

Since Chapter 0 is intended as a summary of the material contained in the complete SPEC book, we present here only references to work related to the particular

versions of the logic and language we use throughout this volume. For references to different, but related, logics, we refer the reader to the SPEC book.

## Fair Transition Systems

*Transition systems* as a generic model for concurrent programs were introduced by Keller [1976]. The basic notion of modeling concurrency by *interleaving* was first used by Dijkstra [1965]. The term “*interleaving*” was coined by Dijkstra [1971]. These references and Dijkstra [1968a] have laid the foundations for the model used throughout this book, emphasizing the interplay between concurrency and *nondeterminism*.

**Fairness:** The simplest notion of fairness, to which we refer as *justice*, is an inevitable logical conclusion of the two premises underlying the interleaving model which were postulated by Dijkstra:

- Nothing may be assumed about the relative speeds of the  $N$  computers; we may not even assume their speed to be constant in time (Dijkstra [1965]).
- “We now request that a process, once initiated, will — sooner or later — get the opportunity to complete its actions” (Dijkstra [1968a]).

One of the first researchers to realize that fairness, which is essential to the abstract representation of concurrency, complicates the mathematical semantics of programs was Park [1980]. He studied fairness in the context of data-flow languages, focusing on the *fair-merge* process as the main implementor of fairness. Related investigations are reported in Park [1981, 1983] and further elaborated by Stomp, de Roever, and Gerth [1989].

A comprehensive survey of the diverse notions of fairness that have been proposed was presented by Francez [1986]. The paper by Apt, Francez and Katz [1988] examines the effectiveness and utility of adopting various notions of fairness. It has been instrumental in our final decision about the fairness notions adopted in the definition of a fair transition system.

**Communication and synchronization:** Of the various mechanisms for communication between processes considered here, *shared variables* is the mechanism used by Dijkstra [1965] for formulating the mutual-exclusion problem. Several variants of *message passing*, including both bounded and unbounded buffering, are considered by Dijkstra [1968a], who also introduced in the same paper synchronization by *semaphores*. *Asynchronous message passing* has been used by Kahn [1974] as the basis for his data-flow model of parallelism.

*Synchronous message passing* in the style used here, where sender and receiver are distinguished, was introduced first by Hoare [1978] in a paper which introduces the language CSP. This mechanism is the one used in the programming language OCCAM. The same mechanism in a more abstract setting, which concentrates on

the synchronization induced by synchronous message passing, was adopted in the language (calculus) CCS as presented in Milner [1980, 1989].

We refer the reader to the comprehensive survey of various modes of communication and synchronization between processes presented in Andrews [1991].

**Programming features:** We can trace the origins of some of the other statements in our programming language. As mentioned above, *semaphores* were first introduced in Dijkstra [1968a], where they were used for solving the mutual-exclusion and other synchronization problems.

The *await* statement was introduced in Lamport [1976]. The *cooperation* statement, allowing a sequential process to spawn several parallel ones in mid-execution, is modeled after the *parbegin* statement of Dijkstra [1968a]. The notion of *guarded command*, which is the basis for the *when* and the *selection* statements, was first introduced by Dijkstra [1975] and further elaborated by Dijkstra [1976].

An important element of the guarded command language of Dijkstra [1976] (see also Gries [1981]) is the nondeterministic choice *if* statement which, in our language, is represented as a *selection* statement with *when* statements as children. In the shared-variables model, this construct is convenient but not absolutely necessary. It can always be programmed by a *while* statement that checks the guards one at a time and executes the statement associated with the first guard found to be true. The situation is different in the message-passing model, where the construct of waiting for an incoming input on one of several channels cannot be programmed without guarded selection. This was first pointed out by Hoare [1978] and is now recognized as one of the most important elements of message-passing languages.

For motivation and detailed discussion of the LCR restriction, we refer readers to the SPEC book.

## Temporal Logic

Temporal logic can be viewed as a special case of *modal logic* which is concerned with the notions of *necessity* and *possibility*. The presently widely accepted possible worlds semantics for modal (and temporal) logics was developed by Kripke [1963]. For comprehensive treatment of modal logic see Hughes and Cresswell [1968] and Chellas [1980].

According to this view, temporal logic can be obtained from modal logic by interpreting the modal operators in a time-dependent context, or, alternatively, by specializing the logic to *time modalities*. This view is presented in Rescher and Urquhart [1971] and Goldblatt [1987]. A classification of various time structures and the axioms that characterize them is presented in Burgess [1984].

Another approach and motivation for the study of temporal logic is through the

logical analysis of natural languages. It views the development of temporal logic as the formalization of linguistic conventions with regard to tenses into a formal calculus. The fundamental observations of McTaggart that stimulated this approach are surveyed in McTaggart [1927]. This approach is fully applied in the monumental work of Prior summarized in Prior [1967]. Other central works that adopt the linguistic view are Kamp [1968], Gabbay [1976], and van Benthem [1983].

**Temporal logic in computer science:** General surveys on the role of temporal logic in computer science include Pnueli [1986], Goldblatt [1987], Emerson [1990], and Stirling [1992].

A reasoning style that closely resembles temporal reasoning was proposed in Burstall [1974] and later developed into the *intermittent assertions* proof method in Manna and Waldinger [1978]. A temporal-like calculus for specification and reasoning about sequential programs was proposed by Kröger [1977]. The application of temporal logic to specification and reasoning about concurrent programs was first proposed in Pnueli [1977], and a temporal semantics for reactive programs was presented in Pnueli [1981] and Manna and Pnueli [1983a, 1991a].

Some of the earlier applications of temporal logic for the specification and verification of concurrent programs are reported by Halpern [1982], Halpern and Owicki [1983], Owicki and Lamport [1982], and Lamport [1983a]. Applications of temporal logic for specifying sequential circuits are proposed in Malachi and Owicki [1981] and Bochman [1982].

**Propositional temporal logic:** The strict *until* and *since* operators were introduced into linear-time temporal logic by Kamp [1968], and shown to be more expressive than  $\Box$  and  $\Diamond$  alone, for both dense and discrete time structures. Kamp also proved the expressive completeness (relative to the first-order monadic theory of linear orders) of full propositional linear-time temporal logic (including the past operators) with both *until* and *since* operators. The expressive completeness of the future fragment alone was obtained by Gabbay, Pnueli, Shelah, and Stavi [1980], together with a complete axiomatic system and a decision procedure for the satisfiability problem. Lichtenstein, Pnueli, and Zuck [1985] points out that past operators prove helpful in compositional reasoning about programs and lead to a more uniform classification of program properties expressible in temporal logic, thus, returning to the full version of temporal logic, as considered by Prior and Kamp. Past operators were also used by Koymans and de Roever [1983] to specify a buffer. The complexity of deciding the satisfiability problem for several propositional temporal logics over discrete models is analyzed in Sistla and Clarke [1985].

**Quantification:** Rigid quantification in temporal logic was proposed in Manna and Pnueli [1981a]. Flexible quantification over propositions is considered by Sistla [1983] and Wolper [1983], and its complexity is analyzed in Sistla, Vardi,

and Wolper [1987]. The general issue of quantification in a modal context is surveyed by Garson [1984] and motivated in Bacon [1980].

**First-order temporal logic:** A proof system for first-order temporal logic is given by Manna and Pnueli [1983c]. Abadi and Manna [1990] consider several proof systems for first-order temporal logic. However, Abadi [1989] showed that first-order temporal logic has no complete axiomatization. The notions of program validity and relative completeness are discussed by Manna and Pnueli [1983a, 1991a].

**Next:** The next operator was introduced as a primitive operator in Manna and Pnueli [1979]. It can be derived from the strict until operator, or alternatively, obtained in terms of the reflexive *until* and flexible quantification. Lamport [1983b] strongly objects to the use of the next operator in a specification language because it is sensitive to stuttering (duplication of a state in a computation). Consequently, he consistently uses a temporal language with reflexive operators and no *next*. The temporal logic of actions TLA proposed in Lamport [1994], reinstates a “next value” operator  $(\cdot)'$  applied to terms (similar to our  $(\cdot)^+$ ) but only allows its use in restricted contexts that ensure insensitivity to stuttering.

**Waiting-for:** The waiting-for (unless) operator, which can be viewed as a weak version of the *until*, is introduced in Manna and Pnueli [1983c], where it is used for a uniform representation of precedence properties.

## Specification

Examples of the use of temporal logic for specifying program properties appear in almost all papers that discuss the logic; samples are Pnueli [1977], Manna and Pnueli [1981a], Manna [1982], Hailpern and Owicki [1983], Hailpern [1982], Owicki and Lamport [1982], Koymans and de Roever [1983], Lamport [1983a], Lamport [1983b], Emerson and Clarke [1982], and many others. In fact, some of these properties have been described in alternative formalisms that predate the use of temporal logic, such as Lamport [1977] and Francez and Pnueli [1978].

Our approach to temporal-logic specifications of message-passing programs is based on Barringer, Kuiper, and Pnueli [1985]. The representation of asynchronous message-passing systems has been influenced by Nguyen, Demers, Owicki, and Gries [1986] and Nguyen, Gries, and Owicki [1985].

**Classification:** Lamport [1977] was the first to classify the properties of reactive systems into the classes of *safety* and *liveness*. A semantic characterization of the safety class was given by Lamport [1985a]. Semantic characterization of the liveness class was provided by Alpern and Schneider [1985]. A syntactic characterization of the safety and liveness properties recognizable by  $\omega$ -automata was given by Alpern and Schneider [1987]. Sistla [1985] provided a full characterization of safety properties specifiable by future propositional temporal logic and a

partial characterization of the liveness properties specifiable in that language.

The safety-progress classification of properties was briefly referred to in Lichtenstein, Pnueli, and Zuck [1985] and fully described by Manna and Pnueli [1990]. The same hierarchy has many characterizations in various formalisms. It has been studied in the context of  $\omega$ -automata by Landweber [1969], Wagner [1979], Arnold [1983], Kaminski [1985], Hoogeboom and Rozenberg [1986], and Staiger [1987].

**Expressiveness:** According to Kamp [1968] and Gabbay, Pnueli, Shelah, and Stavi [1980], both propositional temporal logic and its future fragment have the same expressive power as the first-order monadic theory of linear orders. McNaughton and Papert [1971] establishes an equivalence between this first-order language and several other formalisms in their ability to define languages of finite words. The most important of these formalisms are star-free regular expressions and counter-free automata. These results were extended to show expressive equivalence over infinitary languages by Ladner [1977], Thomas [1979, 1981], Perrin [1984], Arnold [1985], Perrin [1985], and Perrin and Pin [1986]. For a survey of these topics consult Thomas [1990].

Wolper [1983] pointed out that even simple examples, such as the parity property, cannot be specified by unquantified temporal logic, but can be specified by automata. He proceeded to show that when we add to the temporal language grammar operators (or flexible quantification) we obtain a more powerful language which is the same as that of full  $\omega$ -automata or, as shown by Büchi [1960], has the expressive power of the weak second-order monadic theory of a single successor (S1S).

**Compositionality:** The question of compositional specification and verification of concurrent programs has been and still is one of the most active areas of research in the field. It has long been understood that composition of specifications that consist purely of safety properties is easier than composition of general specifications that may contain some liveness properties. References to composition of specifications that may contain liveness include, among others, Misra, Chandy, and Smith [1982], Barringer, Kuiper, and Pnueli [1984], Nguyen, Demers, Owicki, and Gries [1986], Nguyen, Gries, and Owicki [1985], Jonsson [1987], and Abadi and Lamport [1993].

Readers who are interested in compositional verification by methods other than temporal logic are referred to the textbooks by Apt and Olderog [1991] and Francez [1992]. We make no attempt to provide full references to the huge literature devoted to the topic of compositional verification but only mentioned some of the few references that considered compositional verification based on temporal logic.

For more comprehensive discussions of these issues, we refer the readers to the surveys of de Roever [1985] and Hooman and de Roever [1986], and the monograph by Zwiers [1989], covering the field of compositional and modular assertional

specification and verification. They make a distinction between compositional and modular proof approaches, where in the modular approach one must use a fixed specification of the modules to establish all interesting properties of the entire system. According to this distinction, the methods considered in this book are compositional rather than modular.

## Chapter 1

# Invariance: Proof Methods

According to the classification of properties, a *safety property* is a property that can be specified by a formula of the form  $\Box p$  for a past formula  $p$ .

In this chapter, we consider verification of the simplest type of safety properties, those that are specifiable by a formula  $\Box p$ , where  $p$  is a state formula (assertion). For program  $P$ , we refer to such properties as *invariance properties* of  $P$  (*P-invariance properties*), and to the formulas specifying them as *invariance formulas* of  $P$  (*P-invariance formulas*). We say that assertion  $p$  is an *invariant* of  $P$  (*P-invariant*), meaning that  $p$  holds on all  $P$ -accessible states. When  $P$  is clear from the context, we refer to a  $P$ -invariant simply as invariant.

Section 1.1 introduces some preliminary concepts, leading to the verification rule. The basic rule is presented in Section 1.2. A major task in the verification of safety properties is the construction of inductive assertions. In Section 1.3, we present bottom-up methods for such construction, while in Section 1.4 we consider top-down methods.

Section 1.5 points out that many programs can be viewed as a refinement of a more abstract version of the program. Under the assumption that a proof of a safety property for the abstract version is given, the section considers methods for refinement of the proof into a proof that is sound for the refined version of the program.

### 1.1 Preliminary Notions

In preparation for the presentation of proof rules, we introduce some preliminary notions.

## Verification Conditions

An important element that appears in many proof rules is a special formula called a *verification condition*, which expresses the effect of a single transition  $\tau$  by considering two consecutive states connected by  $\tau$ .

Let  $\varphi$  and  $\psi$  be two assertions whose variables are either flexible system variables  $V = Y \cup \{\pi\}$ , consisting of the program variables  $Y$  and the control variable  $\pi$ , or rigid specification variables. The *primed version* of assertion  $\psi$ , denoted by  $\psi'$ , is obtained by replacing each free occurrence of a system variable  $y \in V$  by its primed version  $y'$ .

The *verification condition* (or *proof obligation*) of  $\varphi$  and  $\psi$ , relative to transition  $\tau$ , is given by the state formula

$$\rho_\tau \wedge \varphi \rightarrow \psi'.$$

We adopt the notation

$$\{\varphi\} \tau \{\psi\}$$

as an abbreviation for this verification condition.

**Example** Consider a fair transition system, in which  $V = \{x, y\}$ , and let transition  $\tau$  have the transition relation

$$\rho_\tau: x \geq 0 \wedge y' = x + y \wedge x' = x.$$

Consider the assertions

$$\varphi: y = 3 \quad \text{and} \quad \psi: y = x + 3.$$

Then the verification condition of  $\varphi$  and  $\psi$ , relative to  $\tau$ , is given by the implication

$$\underbrace{x \geq 0 \wedge y' = x + y \wedge x' = x}_{\rho_\tau} \wedge \underbrace{y = 3}_{\varphi} \rightarrow \underbrace{y' = x' + 3}_{\psi'}.$$

Since  $y' = x + y$ ,  $x' = x$ , and  $y = 3$  imply  $y' = x' + 3$ , the verification condition is obviously valid. From this we can conclude that any state, obtained by applying  $\tau$  to a state satisfying  $y = 3$ , will necessarily satisfy  $y = x + 3$ . ■

## Verification Conditions for Modes

Recall that the transition relations of the *conditional* and *while* statements are disjunctions of the form

$$\rho_\tau: \rho_\tau^T \vee \rho_\tau^F.$$

We refer to  $\tau^T$  and  $\tau^F$  (and their associated transition relations  $\rho_\tau^T$  and  $\rho_\tau^F$ ) as the *modes* of transition  $\tau$  (transition relation  $\rho_\tau$ ).

For these transitions, we may form verification conditions for the separate modes. These are

$$\begin{aligned}\{\varphi\} \tau^T \{\psi\}: & \quad \rho_\tau^T \wedge \varphi \rightarrow \psi' \\ \{\varphi\} \tau^F \{\psi\}: & \quad \rho_\tau^F \wedge \varphi \rightarrow \psi'.\end{aligned}$$

It is not difficult to see that  $\{\varphi\} \tau \{\psi\}$  is logically equivalent to the conjunction of the two separate modes, i.e.,

$$\{\varphi\} \tau \{\psi\}: \quad \{\varphi\} \tau^T \{\psi\} \wedge \{\varphi\} \tau^F \{\psi\}.$$

Thus, to establish the validity of  $\{\varphi\} \tau \{\psi\}$  it is sufficient to establish separately the validity of  $\{\varphi\} \tau^T \{\psi\}$  and  $\{\varphi\} \tau^F \{\psi\}$ .

**Example** Consider the statement

$$\ell: \text{if } x > 0 \text{ then } \ell_1: y := 1 \text{ else } \ell_2: y := 2,$$

and the assertions

$$\varphi: T \quad \text{and} \quad \psi: at\_{\ell_1} \vee at\_{\ell_2}.$$

Recall (page 18) that  $at\_{\ell}$  is defined as  $[\ell] \in \pi$ .

To establish  $\{\varphi\} \ell \{\psi\}$ , it is sufficient to prove separately

$$\underbrace{\cdots \wedge x > 0 \wedge move(\ell, \ell_1) \wedge \cdots}_{\rho_\ell^T} \wedge \underbrace{\varphi}_{\varphi} \rightarrow \underbrace{at'\_{\ell_1} \vee at'\_{\ell_2}}_{\psi'}$$

and

$$\underbrace{\cdots \wedge x \leq 0 \wedge move(\ell, \ell_2) \wedge \cdots}_{\rho_\ell^F} \wedge \underbrace{\varphi}_{\varphi} \rightarrow \underbrace{at'\_{\ell_1} \vee at'\_{\ell_2}}_{\psi'}.$$

Recall (page 18) that  $at'\_{\ell}$  is defined as  $[\ell] \in \pi'$ , and  $move(\ell, \ell_1)$  is defined as (page 19):

$$[\ell] \in \pi \wedge \pi' = (\pi - \{[\ell]\}) \cup \{[\ell_1]\}. \quad \blacksquare$$

## Leads From-To

Let  $\varphi$  and  $\psi$  be two assertions.

- For a transition  $\tau$  in system  $P$ , we say that  $\tau$  leads from  $\varphi$  to  $\psi$  in  $P$ , if the verification condition

$$\{\varphi\} \tau \{\psi\}$$

holds.

- For a set of transitions  $T \subseteq \mathcal{T}$ , where  $\mathcal{T}$  is the set of all transitions in  $P$ , we say that  $T$  leads from  $\varphi$  to  $\psi$  in  $P$ , and write

$$\{\varphi\} T \{\psi\},$$

if  $\{\varphi\} \tau \{\psi\}$  holds for every transition  $\tau \in T$ .

We refer to triplets such as

$$\{\varphi\} \tau \{\psi\} \quad \{\varphi\} T \{\psi\} \quad \text{and} \quad \{\varphi\} P \{\psi\}$$

as the *verification conditions* for the transition  $\tau$ , the set of transitions  $T$ , and the complete program  $P$ , respectively.

## Predictive Computations

To establish the properties of verification conditions, we introduce the notion of a predictive computation.

A model (sequence of states) is called *predictive* if each state  $s_j$  in the model interprets each primed version  $x'$  of a system variable  $x \in V$  in the same way that  $s_{j+1}$  interprets  $x$ . That is,  $s_j[x'] = s_{j+1}[x]$ , for each  $x \in V$  and every  $j = 0, 1, \dots$ . A computation is called *predictive* if it is a predictive model.

For example, the following computation is predictive for the case that  $V = \{x\}$ :

$$\sigma_1: \langle x: 0, x': 1 \rangle, \langle x: 1, x': 2 \rangle, \langle x: 2, x': 3 \rangle, \dots .$$

Obviously, for every computation  $\sigma$  of  $P$ , there exists a predictive model  $\tilde{\sigma}$  which agrees with  $\sigma$  on the interpretation of the system variables  $V$  (but not necessarily on the interpretation of their primed version  $V'$ ). Since the conditions for being a computation only depend on the interpretation of the system variables  $V$ ,  $\tilde{\sigma}$  is also a computation. We refer to  $\tilde{\sigma}$  as the *predictive version* of  $\sigma$ . For example,  $\sigma_1$  above is the predictive version of the computation

$$\sigma_2: \langle x: 0, x': 0 \rangle, \langle x: 1, x': 1 \rangle, \langle x: 2, x': 2 \rangle, \dots .$$

An important property of a predictive computation is that if  $s_{j+1}$  is a  $\tau$ -successor of  $s_j$  then  $\rho_\tau(V, V')$  holds at position  $j$ .

## Consequences of a Valid Verification Condition

The main role of the verification condition is to establish that every  $\tau$ -successor of a  $\varphi$ -state in a computation of the program  $P$  is a  $\psi$ -state. This is stated by the following claim:

### Claim (verification condition)

If  $\{\varphi\} \tau \{\psi\}$  is  $P$ -state valid, then every  $\tau$ -successor of a  $P$ -accessible  $\varphi$ -state is a  $\psi$ -state.

**Justification** Assume that  $\{\varphi\} \tau \{\psi\}$  is  $P$ -state valid and let  $\sigma: s_0, s_1, \dots$  be a computation of  $P$  such that  $s_i$  is a  $\varphi$ -state and  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ . We show that  $s_{i+1}$  is a  $\psi$ -state.

Let  $\tilde{\sigma}: \tilde{s}_0, \tilde{s}_1, \dots$  be a predictive version of  $\sigma$ . Since  $\tilde{\sigma}$  is a computation and  $\{\varphi\} \tau \{\psi\}$  is  $P$ -state valid, state  $\tilde{s}_i$  satisfies  $\{\varphi\} \tau \{\psi\}: \rho_\tau \wedge \varphi \rightarrow \psi'$ . As the states

of  $\tilde{\sigma}$  agree with their corresponding states in  $\sigma$  on the interpretation of  $V$  and  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ ,  $\tilde{s}_{i+1}$  is a  $\tau$ -successor of  $\tilde{s}_i$ , and hence state  $\tilde{s}_i$  satisfies  $\rho_\tau$ . As  $\varphi$  depends only on the (unprimed) system variables  $V$  and states  $s_i$  and  $\tilde{s}_i$  agree on their interpretation,  $\tilde{s}_i$  also satisfies  $\varphi$ .

It follows that  $\tilde{s}_i$  satisfies  $\rho_\tau \wedge \varphi \rightarrow \psi'$ ,  $\rho_\tau$ , and  $\varphi$ , implying that it satisfies  $\psi'$ , that is  $\tilde{s}_i[\psi'] = \top$ . As the computation is predictive,  $\tilde{s}_{i+1}[V] = \tilde{s}_i[V']$  and therefore  $\tilde{s}_{i+1}[\psi] = \tilde{s}_i[\psi'] = \top$ . It follows that  $\tilde{s}_{i+1}$  satisfies  $\psi$  and, since  $s_{i+1}$  agrees with  $\tilde{s}_{i+1}$  on  $V$ , also  $s_{i+1}$  satisfies  $\psi$ . ■

In Problem 1.1, we ask the reader to prove the converse of this claim. That is, if every  $\tau$ -successor of a  $P$ -accessible  $\varphi$ -state is a  $\psi$ -state then the verification condition  $\{\varphi\} \tau \{\psi\}$  is  $P$ -state valid.

## Observations about Verification Conditions

As we will see below, most of our proof rules are based on verification conditions, deducing the  $P$ -validity of a temporal formula from the  $P$ -state validity of assertional premises. Thus, we may regard the verification conditions as the building blocks of proofs of temporal properties. Consequently, we should acquire good techniques for proving verification conditions.

In order to concentrate on the temporal aspects of proofs, we will rarely provide a detailed proof of the general state validity on which we rely. When necessary, we will provide informal arguments for the less obvious cases. In a later section, we will also suggest some heuristics for quick identification of transitions in  $T$  for which the verification condition is trivially valid.

There are some general considerations that simplify the proofs of verification conditions. We review some of them here.

### Inferring $P$ -State Validity from State Validity

The simplest and most frequently used method for establishing the  $P$ -state validity of a verification condition is to infer it from the general state validity of the same condition. This is, of course, no more than a special case of the general observation made in Chapter 0, by which an assertion that is state valid is also  $P$ -state valid for any program  $P$ .

For example, the  $P$ -state validity of the verification condition

$$\underbrace{y' = x + y \wedge x' = x}_{\rho_\tau} \wedge \underbrace{y = 3}_{\varphi} \rightarrow \underbrace{y' = x' + 3}_{\psi'}$$

is easily provable by observing that it is generally state valid.

## Simplification by Partial Substitution

Most transition relations have the general form

$$\rho_\tau: C_\tau \wedge \bar{y}' = \bar{e} \wedge \text{pres}(\bar{v}),$$

in which the system variables  $V$  are partitioned into the  $\tau$ -modifiable variables  $\bar{y}$  and the  $\tau$ -preserved variables  $\bar{v}$ . Substituting only for the  $\tau$ -modifiable variables, we obtain the following *simplified version* of the *verification condition*:

$$\{\varphi\} \tau \{\psi\}: C_\tau \wedge \bar{y}' = \bar{e} \wedge \varphi \rightarrow \psi[\bar{y} \mapsto \bar{y}'].$$

The notation  $\psi[\bar{y} \mapsto \bar{y}']$  denotes a version of the assertion  $\psi$  in which each occurrence of  $y_i$  (assuming  $\bar{y} = (y_1, \dots, y_m)$ ) is replaced by  $y'_i$ .

For example, the simplified version of the verification condition

$$y' = x + y \wedge x' = x \wedge y = 3 \rightarrow y' = x' + 3$$

is

$$y' = x + y \wedge y = 3 \rightarrow y' = x + 3.$$

From now on, whenever possible, we will use the simplified versions of the verification conditions. We still refer to  $\psi[\bar{y} \mapsto \bar{y}']$  as  $\psi'$  even though it contains unprimed versions of the preserved variables.

## Simplification of Control Expressions

In a verification condition of the form

$$\rho_\tau \wedge \varphi \rightarrow \psi',$$

it is possible to use logical consequences of  $\rho_\tau$  to simplify control expressions in  $\varphi$  and in  $\psi'$ . Here, we explore the implications of the conjunct  $\text{move}(L_1, L_2)$  appearing in most transition relations  $\rho_\tau$ . Recall that the formula  $\text{move}(L_1, L_2)$  is an abbreviation for

$$\text{move}(L_1, L_2): L_1 \subseteq \pi \wedge \pi' = (\pi - L_1) \cup L_2.$$

Following are some of the consequences of the formula  $\text{move}(L_1, L_2)$ . These consequences can be used to simplify assertions  $\varphi$  and  $\psi'$  of the verification condition.

- $\text{at\_}\ell_1 = \top$  for every  $[\ell_1] \in L_1$
- $\text{at}'_{\ell_2} = \top$  for every  $[\ell_2] \in L_2$
- $\text{at}'_{\ell_3} = \text{f}$  for every  $[\ell_3] \in L_1 - L_2$
- $\text{at}'_m = \text{at\_}_m$  for every  $[m] \notin L_1 \cup L_2$ .

In later discussions we justify the use of these consequences by saying that they are implied by  $\text{move}(L_1, L_2)$  (or any transition relation  $\rho_\tau$  that contains the conjunct  $\text{move}(L_1, L_2)$ ).

### The Idling Transition

Since the idling transition  $\tau_I$  is present in all transition systems corresponding to programs, we are repeatedly called upon to establish the validity of the verification condition

$$\rho_{\tau_I} \wedge \varphi \rightarrow \varphi'.$$

This formula is always valid for an assertion  $\varphi$  that only refers to system or rigid variables. This is because  $\tau_I$  modifies no system or rigid variables and therefore  $\varphi' = \varphi$ , leading to the obviously valid implication

$$T \wedge \varphi \rightarrow \varphi.$$

Consequently, all assertions  $\varphi$  are preserved by the idling transition, and from now on we will never consider it when establishing  $\{\varphi\} T \{\varphi\}$ .

## 1.2 Invariance Rule

In this section, we introduce several rules for proving the  $P$ -validity of invariance formulas.

### The Basic Rule

The simplest rule for proving invariance properties is rule INV-B (Fig. 1.1), which establishes an assertion  $\varphi$  as an invariant of a program  $P$  ( $P$ -invariant), i.e.,  $\Box \varphi$  is  $P$ -valid.

For an assertion  $\varphi$ ,

$$\frac{\begin{array}{l} B1. \quad \Theta \rightarrow \varphi \\ B2. \quad \{\varphi\} T \{\varphi\} \end{array}}{\Box \varphi}$$

Fig. 1.1. Rule INV-B (basic invariance).

Premise B1 of rule INV-B requires that  $\Theta$ , the initial condition of  $P$ , implies  $\varphi$ , i.e.,  $\Theta \rightarrow \varphi$  is  $P$ -state valid. Premise B2 of the rule,  $\{\varphi\} T \{\varphi\}$ , requires that all transitions in  $P$  preserve  $\varphi$ , i.e.,

$$\rho_T \wedge \varphi \rightarrow \varphi'$$

is  $P$ -state valid for every  $\tau \in T$ . Together, the two premises ensure that for every computation of  $P$ ,  $\varphi$  holds initially and is preserved by every transition. Consequently,  $\varphi$  must hold in all states of every computation as stated by the conclusion of the rule.

Note that, according to our convention, the two premises claim the  $P$ -state validity of the respective implications, while the conclusion claims the (temporal  $P$ -validity of  $\Box \varphi$ . Also observe that, for an assertion  $\varphi$ ,  $\Box \varphi$  being  $P$ -valid is the same as  $\varphi$  being  $P$ -state valid.

Thus, the full-form presentation of rule INV-B, explicitly displaying the type of validity assumed on each line is given by

$$\text{B1. } P \Vdash \Theta \rightarrow \varphi$$

$$\text{B2. } P \Vdash \rho_\tau \wedge \varphi \rightarrow \varphi' \text{ for each } \tau \in T$$

$$\underline{P \models \Box \varphi}$$

The soundness of rule INV-B can be justified by induction on the position of a state in a computation. Consider a computation

$$\sigma: s_0, s_1, \dots$$

Premise B1 ensures that  $s_0$  satisfies  $\varphi$ . Premise B2 ensures, for every  $i = 0, 1, \dots$  that if  $s_i$  satisfies  $\varphi$ , then so does  $s_{i+1}$ . It follows by induction that  $s_k$  satisfies the assertion  $\varphi$  for every  $k \geq 0$ .

In Problem 1.2 we request the reader to prove the converse direction of rule INV-B. That is, for program  $P$ , if assertion  $\varphi$  is  $P$ -invariant, then the two premises B1 and B2, are  $P$ -state valid.

### Example (add-two)

Consider program ADD-TWO (Fig. 1.2) which adds the constant 2 to variable  $x$ . We wish to prove for this program the invariance of the assertion

$$\varphi: x \geq 0.$$

```
local x: integer where x = 0
```

```
l0: x := x + 2
```

```
l1:
```

Fig. 1.2. Program ADD-TWO.

To do so, we establish separately the two premises of rule INV-B.

- Establishing  $\Theta \rightarrow \varphi$ :

Premise B1 is

$$\underbrace{\pi = \{\ell_0\} \wedge x = 0}_{\Theta} \rightarrow \underbrace{x \geq 0}_{\varphi},$$

which is obviously state valid.

- Establishing  $\{\varphi\} T \{\varphi\}$ :

Having observed that the idling transition  $\tau_i$  preserves all assertions, we consider the remaining transition  $\ell_0$ .

The verification condition for  $\ell_0$  is

$$\underbrace{\dots \wedge x' = x + 2}_{\rho_{\ell_0}} \wedge \underbrace{x \geq 0}_{\varphi} \rightarrow \underbrace{x' \geq 0}_{\varphi'},$$

which is also state valid. By rule INV-B, we conclude the  $P$ -validity of the invariant  $\square(x \geq 0)$ . ■

## Monotonicity and Conjunctiveness

There are several rules that enable us to derive new invariance formulas from previously established invariances.

Rule MON-I (Fig. 1.3) states, for assertions  $p$  and  $q$ , that if  $p$  is  $P$ -invariant and  $p \rightarrow q$  is state valid, then  $q$  is also  $P$ -invariant. We may use rule MON-I to infer the invariance  $\square(x > -1)$  for program ADD-TWO from the previously established invariance  $\square(x \geq 0)$  and the state-valid implication  $x \geq 0 \rightarrow x > -1$ .

For assertions  $p$  and  $q$ ,

$$\frac{\square p, \quad p \rightarrow q}{\square q}$$

Fig. 1.3. Rule MON-I (monotonicity of invariances).

The next rule, rule CON-I of Fig. 1.4, states that if assertions  $p$  and  $q$  are both  $P$ -invariant then so is  $p \wedge q$ . Note that the converse of the rule is also sound. That is, if  $p \wedge q$  is  $P$ -invariant then so are  $p$  and  $q$ . This can be derived from MON-I and the observation that  $p \wedge q$  implies both  $p$  and  $q$ .

For assertions  $p$  and  $q$ ,

$$\frac{\square p, \quad \square q}{\square(p \wedge q)}$$

Fig. 1.4. Rule CON-I (conjunction of invariances).

Let  $\text{even}(x)$  be a predicate characterizing the even integers. We may use rule CON-I to infer the invariance  $\square(x \geq 0 \wedge \text{even}(x))$  for program ADD-TWO from the previously established invariance  $\square(x \geq 0)$  and from the invariance  $\square(\text{even}(x))$  which can be established in a similar way.

### **Limitations of the Basic Rule**

One of the major problems in verification is that we often wish to prove that  $p$  is an invariant of program  $P$ , i.e.,  $\square p$  is  $P$ -valid, but find that rule INV-B is not directly applicable.

Consider, for example, the assertion

$$\varphi: \text{at-}\ell_1 \rightarrow x = 2$$

for program ADD-TWO (Fig. 1.2). This assertion is obviously an invariant of ADD-TWO. When we attempt to verify it, using rule INV-B, we have to establish the two implications

$$\text{B1: } \underbrace{\pi = \{\ell_0\} \wedge x = 0}_{\Theta} \rightarrow \underbrace{\text{at-}\ell_1 \rightarrow x = 2}_{\varphi}$$

$$\text{B2: } \underbrace{\text{move}(\ell_0, \ell_1) \wedge x' = x + 2}_{P_{\ell_0}} \wedge \underbrace{\text{at-}\ell_1 \rightarrow x = 2}_{\varphi} \rightarrow \underbrace{\text{at}'\ell_1 \rightarrow x' = 2}_{\varphi'}$$

Implication B1 is state valid since  $\pi = \{\ell_0\}$  implies  $\neg \text{at-}\ell_1$ .

In implication B2,  $\text{move}(\ell_0, \ell_1)$  implies  $\text{at}'\ell_1$ . Consequently, B2 can be simplified to  $\text{at-}\ell_0 \wedge (\text{at-}\ell_1 \rightarrow x = 2) \rightarrow (x + 2 = 2)$  which, however, is not state valid.

Since the only method presented so far for proving the  $P$ -state validity of an implication, such as  $\text{at-}\ell_0 \rightarrow x = 0$ , is to infer it from a state validity, rule INV-B is not directly applicable. Note, however, that the implication  $\text{at-}\ell_0 \rightarrow x = 0$  is  $P$ -state valid. That is, all states that occur in computations of ADD-TWO satisfy  $\text{at-}\ell_0 \rightarrow x = 0$ . Thus, the difficulty can be traced to the problem of proving the  $P$ -state validity of implications that are  $P$ -state valid, but not state valid.

We present two solutions to this problem, which can be summarized by the following strategies:

1. Use a stronger assertion, or
2. Conduct an incremental proof, using previously established  $P$ -invariants.

In the following, we will elaborate these two strategies.

### **Strategy 1: Using a Stronger Assertion**

An assertion  $\varphi$  is called *inductive* if the two premises B1 and B2 of rule INV-B are state valid. Obviously, if premises B1 and B2 are state valid, they are also  $P$ -state valid and therefore, by rule INV-B,  $\varphi$  is  $P$ -invariant. We conclude that every inductive assertion is  $P$ -invariant. The example we have just considered shows that the converse of this statement is not always true. Assertion  $at\_l_1 \rightarrow x = 2$  is  $P$ -invariant but not inductive.

This situation is familiar to anyone who has ever used mathematical induction to prove simple arithmetical statements. For example, the statement

“ $1 + 3 + 5 + \cdots + (2k + 1)$  is the square of an integer”

is true but cannot be used as the induction hypothesis in a proof by mathematical induction on  $k$ . The standard remedy to such situations is to find a *stronger* statement that is inductive and implies the desired statement. For example, the stronger statement

$$1 + 3 + 5 + \cdots + (2k + 1) = (k + 1)^2$$

can be used as the induction hypothesis and implies that  $1 + 3 + \cdots + (2k + 1)$  is the square of an integer.

By analogy, to prove that assertion  $p$  is  $P$ -invariant while it is not inductive, we look for an assertion  $\varphi$  that is stronger than  $p$  (i.e.,  $\varphi$  implies  $p$ ) and is inductive. Inductiveness of the *strengthened* assertion  $\varphi$  implies  $\square \varphi$  and, since  $\varphi \rightarrow p$ ,  $\square p$  follows by rule MON-I.

This approach is summarized by rule INV (Fig. 1.5), which can be viewed as a combination of rules INV-B and MON-I. Premises I2 and I3 of INV correspond to premises B1 and B2 of INV-B, from which we can infer  $\square \varphi$ . The implication  $\varphi \rightarrow p$  given in premise I1 yields  $\square p$  by MON-I.

Rule INV claims that if the implications listed in premises I1–I3 are  $P$ -state valid, then  $\square p$  is  $P$ -valid, i.e.,  $p$  is a  $P$ -invariant.

The strategy associated with rule INV can be stated as:

To prove  $\square p$ , find an inductive assertion  $\varphi$  that is stronger than  $p$ .

It is important to realize that if  $p$  is a  $P$ -invariant, then there always exists an

For assertions  $\varphi, p$ ,

$$\frac{\begin{array}{l} \text{I1. } \varphi \rightarrow p \\ \text{I2. } \Theta \rightarrow \varphi \\ \text{I3. } \{\varphi\} T \{\varphi\} \end{array}}{\square p}$$

Fig. 1.5. Rule INV (general invariance).

inductive assertion  $\varphi$  stronger than  $p$ . This will be proven in Section 2.5.

### Example (add-two)

Consider again program ADD-TWO (Fig. 1.2) in which  $x$  is incremented by 2. We wish to prove the partial correctness of this program (see page 62) with respect to the postcondition  $x = 2$ ; that is, we wish to show that on termination  $x = 2$ . Stated as a safety property, this is expressible as the invariance of the assertion

$$p: \text{at-}\ell_1 \rightarrow x = 2.$$

This assertion is obviously an invariant of the program.

As demonstrated in our previous attempt to verify this invariance, assertion  $p$  is not inductive because the verification condition

$$\{p\} \ell_0 \{p\}: \rho_{\ell_0} \wedge \underbrace{\text{at-}\ell_1 \rightarrow x = 2}_{p} \rightarrow \underbrace{\text{at}'_{\ell_1} \rightarrow x' = 2}_{p'}$$

is not state valid. Therefore, rule INV-B is not directly applicable for proving this invariance.

The recommended remedy to such situations is to strengthen  $p$  into an inductive assertion  $\varphi$  and apply rule INV. Since the obstacle to inductiveness of  $p$  is that the implication  $\text{at-}\ell_0 \rightarrow x = 0$  is not state valid, we employ a heuristic that succeeds in a large number of cases and strengthen  $p$  by adding the offending implication as a conjunct, thus transforming an obstacle into an advantage. This leads to the assertion

$$\varphi: (\text{at-}\ell_1 \rightarrow x = 2) \wedge (\text{at-}\ell_0 \rightarrow x = 0).$$

Let us show that  $\varphi$  satisfies the premises of rule INV as state validities.

Premise I1 is given by the implication

$$\underbrace{(\text{at-}\ell_1 \rightarrow x = 2) \wedge (\text{at-}\ell_0 \rightarrow x = 0)}_{\varphi} \rightarrow \underbrace{\text{at-}\ell_1 \rightarrow x = 2}_{p}$$

which is trivially state valid.

Premise I2 is

$$\underbrace{\pi = \{\ell_0\} \wedge x = 0}_{\Theta} \rightarrow \underbrace{(at_{-\ell_1} \rightarrow x = 2) \wedge (at_{-\ell_0} \rightarrow x = 0)}_{\varphi}.$$

Since  $\Theta$  implies  $\neg at_{-\ell_1}$ ,  $at_{-\ell_0}$ , and  $x = 0$ , the implication is state valid.

Premise I3 leads to

$$\underbrace{move(\ell_0, \ell_1) \wedge x' = x + 2}_{\rho_{\ell_0}} \wedge \underbrace{(at_{-\ell_1} \rightarrow x = 2) \wedge (at_{-\ell_0} \rightarrow x = 0)}_{\varphi} \rightarrow \underbrace{(at'_{-\ell_1} \rightarrow x' = 2) \wedge (at'_{-\ell_0} \rightarrow x' = 0)}_{\varphi'}.$$

Since  $move(\ell_0, \ell_1)$  implies  $at_{-\ell_0}$ ,  $\neg at'_{-\ell_0}$ , and  $at'_{-\ell_1}$ , the implication can be simplified to

$$\dots \wedge x' = x + 2 \wedge x = 0 \rightarrow x' = 2,$$

which is obviously state valid.

Note that assertion  $\varphi$ , just shown to be inductive, is a conjunction of the two assertions:  $at_{-\ell_1} \rightarrow x = 2$  and  $at_{-\ell_0} \rightarrow x = 0$ , neither of which is inductive when standing alone. ■

We illustrate the need for strengthening assertions with an additional example.

### Example (mutual exclusion by semaphores)

Consider program MUX-SEM of Fig. 1.6 (see also Fig. 0.7, page 30), which achieves mutual exclusion by semaphores.

local  $y$ : integer where  $y = 1$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } y \\ \ell_3: \text{critical} \\ \ell_4: \text{release } y \end{array} \right] \end{array} \right]$	$\parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \text{request } y \\ m_3: \text{critical} \\ m_4: \text{release } y \end{array} \right] \end{array} \right]$
--	---

Fig. 1.6. Program MUX-SEM (mutual exclusion by semaphores).

The main safety property we wish to prove for this program is mutual exclusion, that is, the invariance of

$$p: \neg(at\_l_3 \wedge at\_m_3).$$

This assertion provides information only about states in which either  $at\_l_3$  or  $at\_m_3$  is true. It is not an inductive assertion, as can be seen by considering the verification condition with respect to  $\ell_2$ .

Let us consider an inductive assertion  $\varphi$ , stronger than  $p$ , which provides more information about the intermediate states.

A state is said to be *critical* for  $p$  if it satisfies  $p$  but is one transition before a state that may potentially violate  $p$ . Such is, for example, the state in which  $\pi = \{\ell_2, m_3\}$ . It satisfies  $p$ , but if the transition  $\ell_2$  could be performed it would lead to the state  $\pi = \{\ell_3, m_3\}$ , which violates  $p$ . We conclude that while control is at  $m_3$ , the transition  $\ell_2$  should be disabled, which is ensured if  $y \leq 0$ . Observing that  $y$  is only modified by the semaphore statements **request** and **release**, and has a nonnegative initial value, it must be invariantly nonnegative. This leads to the inevitable conclusion that while control is at  $\{\ell_2, m_3\}$ , we should have  $y = 0$ . It is easy to see that the data requirement  $y = 0$  can be extended also to states satisfying  $at\_l_{0,1,2} \wedge at\_m_{3,4}$ , i.e., states in which  $P_1$  is at one of the locations  $\{\ell_0, \ell_1, \ell_2\}$  while  $P_2$  is at locations  $m_3$  or  $m_4$ . This is because the transitions departing from locations  $\ell_0$ ,  $\ell_1$ , and  $m_3$  are always enabled when control is in front of their statements, and their execution does not modify  $y$ .

It follows that we can partition the state space of program MUX-SEM of Fig. 1.6 into four subsets, as shown in Fig. 1.7.

To write an inductive assertion that captures the set of states presented in Fig. 1.7, we introduce some additional notation. We write  $\pi_\ell$  and  $\pi_m$  to denote the respective sets of  $\ell_i$ -locations and  $m_i$ -locations currently contained in  $\pi$ . That is,  $\pi_\ell$  and  $\pi_m$  are given by

$$\pi_\ell: \pi \cap \{\ell_0, \dots, \ell_4\} \quad \text{and} \quad \pi_m: \pi \cap \{m_0, \dots, m_4\}.$$

We also allow an extended syntax in which formulas may appear within arithmetic expressions, interpreting T as 1 and F as 0. We refer to this interpretation as the *arithmetization of boolean values*. Using this extension we can write an assertion  $\varphi$ , characterizing all the reachable states appearing in Fig. 1.7:

$$\varphi: |\pi_\ell| = |\pi_m| = 1 \wedge y \geq 0 \wedge at\_l_{3,4} + at\_m_{3,4} + y = 1$$

The first conjunct, which is an abbreviation for  $|\pi_\ell| = 1 \wedge |\pi_m| = 1$ , requires that  $\pi$  contain precisely one  $\ell_i$ -location and one  $m_i$ -location. All three conjuncts are necessary in order to exclude states not appearing in the diagram of Fig. 1.7. For example, omission of the  $|\pi_\ell| = |\pi_m| = 1$  conjunct would allow unwanted states such as  $\langle \pi: \{\ell_0, \ell_3, \ell_4, m_0\}, y: 0 \rangle$ . Similarly, omission of the  $y \geq 0$  conjunct allows unwanted states such as  $\langle \pi: \{\ell_3, m_3\}, y: -1 \rangle$ .

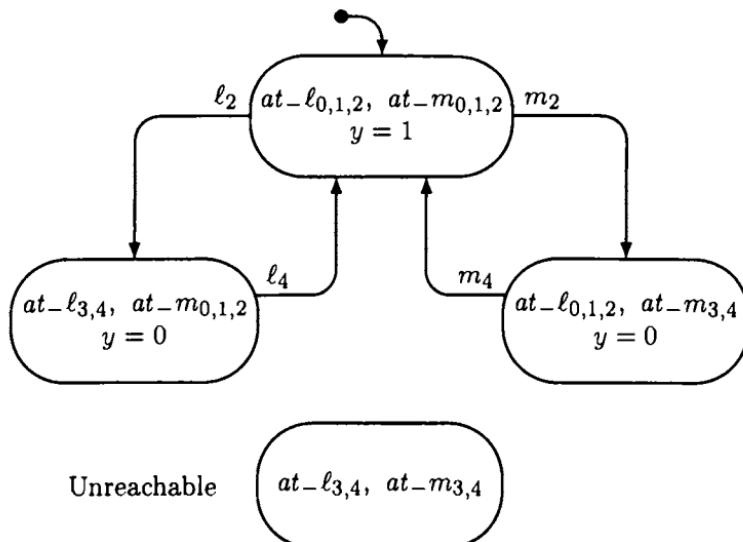


Fig. 1.7. State-space partition (program MUX-SEM).

Having derived a candidate for assertion  $\varphi$ , we use rule INV to prove the invariance of  $p$ :  $\neg(at_{\ell_3} \wedge at_{m_3})$ .

- Premise I1:  $\varphi \rightarrow p$

Obviously, the implication

$$\underbrace{|\pi_\ell| = |\pi_m| = 1 \wedge y \geq 0 \wedge at_{\ell_{3,4}} + at_{m_{3,4}} + y = 1}_{\varphi} \rightarrow \underbrace{\neg(at_{\ell_3} \wedge at_{m_3})}_{p}$$

is valid, since by the left-hand side,  $at_{\ell_3} \wedge at_{m_3}$  implies  $y = -1$ , which contradicts  $y \geq 0$ .

- Premise I2:  $\Theta \rightarrow \varphi$

We have to show

$$\underbrace{\pi = \{\ell_0, m_0\} \wedge y = 1}_{\Theta} \rightarrow \underbrace{|\pi_\ell| = |\pi_m| = 1 \wedge y \geq 0 \wedge at_{\ell_{3,4}} + at_{m_{3,4}} + y = 1}_{\varphi}$$

Clearly,  $\pi = \{\ell_0, m_0\}$  implies  $|\pi_\ell| = |\pi_m| = 1$ . The conjunct  $y = 1$  implies  $y \geq 0$ .

By  $\Theta$ ,  $at_{-\ell_{3,4}} = at_{-m_{3,4}} = 0$  and  $y = 1$ . Hence, we have to show

$$\Theta \rightarrow 0 + 0 + 1 = 1,$$

which is obviously valid.

- Premise I3:  $\{\varphi\} T \{\varphi\}$

Next, we have to show that  $\{\varphi\} \tau \{\varphi\}$  holds for all the diligent (non-idling) transitions in program MUX-SEM. To establish the state validity of the verification condition

$$\rho_T \wedge \underbrace{|\pi_\ell| = |\pi_m| = 1 \wedge y \geq 0 \wedge at_{-\ell_{3,4}} + at_{-m_{3,4}} + y = 1}_{\varphi} \rightarrow$$

$$\underbrace{|\pi'_\ell| = |\pi'_m| = 1 \wedge y' \geq 0 \wedge at'_{-\ell_{3,4}} + at'_{-m_{3,4}} + y' = 1}_{\varphi'}$$

it is sufficient to show that each transition preserves each of the three conjuncts in  $\varphi$ . That is, it is sufficient to establish the state validity of the three following implications for each diligent transition:

$$V_1: \rho_T \wedge |\pi_\ell| = |\pi_m| = 1 \rightarrow |\pi'_\ell| = |\pi'_m| = 1$$

$$V_2: \rho_T \wedge y \geq 0 \rightarrow y' \geq 0$$

$$V_3: \rho_T \wedge |\pi_\ell| = |\pi_m| = 1 \wedge at_{-\ell_{3,4}} + at_{-m_{3,4}} + y = 1 \rightarrow at'_{-\ell_{3,4}} + at'_{-m_{3,4}} + y' = 1.$$

Let us first consider the five transitions of  $P_1$ , corresponding to statements  $\ell_0, \dots, \ell_4$ . The effect of these transitions on  $\pi$  is to replace one  $\ell$ -location by another. Consequently, each of these transitions preserves the values of  $|\pi_\ell|$  and  $|\pi_m|$  and, therefore, preserves the truth of the conjunct  $|\pi_\ell| = |\pi_m| = 1$ .

Next, consider the effect of these five transitions on  $y$ . Transitions  $\ell_0, \ell_1$ , and  $\ell_3$  do not modify  $y$ . Consequently, they preserve the truth of  $y \geq 0$ . For transition  $\ell_2$ ,  $\rho_{\ell_2}$  implies  $y > 0$  and  $y' = y - 1$  which, obviously, leads to  $y' \geq 0$ . For transition  $\ell_4$ ,  $\rho_{\ell_4}$  implies  $y' = y + 1$  which, together with  $y \geq 0$ , leads to  $y' \geq 0$ .

It remains to check that all five transitions preserve the truth of the assertion  $at_{-\ell_{3,4}} + at_{-m_{3,4}} + y = 1$ . We list below the corresponding five implications in which we have already incorporated some simplifications implied by the conjunction  $\rho_{\ell_i} \wedge |\pi_\ell| = |\pi_m| = 1$  and, therefore, can omit this conjunction from the left-hand side of the implications. In particular, under any  $\rho_{\ell_i}$ , we can replace  $at'_{-m_{3,4}}$  by  $at_{-m_{3,4}}$ . The other simplifications determine the values of  $at_{-\ell_{3,4}}$  and  $at'_{-\ell_{3,4}}$  based on  $\rho_{\ell_i} \wedge |\pi_\ell| = 1$ .

- $$\begin{aligned}\ell_0: \quad 0 + at\_m_{3,4} + y = 1 &\rightarrow 0 + at\_m_{3,4} + y = 1 \\ \ell_1: \quad 0 + at\_m_{3,4} + y = 1 &\rightarrow 0 + at\_m_{3,4} + y = 1 \\ \ell_2: \quad 0 + at\_m_{3,4} + y = 1 &\rightarrow 1 + at\_m_{3,4} + (y - 1) = 1 \\ \ell_3: \quad 1 + at\_m_{3,4} + y = 1 &\rightarrow 1 + at\_m_{3,4} + y = 1 \\ \ell_4: \quad 1 + at\_m_{3,4} + y = 1 &\rightarrow 0 + at\_m_{3,4} + (y + 1) = 1.\end{aligned}$$

The implications corresponding to all of these transitions are trivially valid.

In the same way, the five transitions of  $P_2$  can also be shown to preserve  $\varphi$ . This establishes premise I3 of rule INV.

By rule INV, we conclude the invariance of  $p$ :  $\neg(at\_\ell_3 \wedge at\_m_3)$ . ■

## Strategy 2: Incremental Proofs

In some cases it is possible to first prove a simpler invariant by rule INV-B and then use it to establish a more complicated invariant. This is the basis for the second strategy for overcoming the limitations of rule INV-B: using previously established  $P$ -invariants to prove the  $P$ -state validity of assertions.

The main additional tool we use is rule sv-PSV presented in Fig. 1.8, which shows how to derive a  $P$ -state validity of the form  $p \rightarrow q$  from a weaker state validity  $\chi \wedge p \rightarrow q$  plus a  $P$ -invariant  $\chi$ .

For assertions  $p$ ,  $q$ , and  $\chi$ ,

$$\frac{P \models \square \chi, \quad \models \chi \wedge p \rightarrow q}{P \Vdash p \rightarrow q}$$

Fig. 1.8. Rule sv-PSV (from state validities to  $P$ -state validities).

The rule assumes that  $\chi$  has been previously proven to be  $P$ -invariant and that the augmented implication  $\chi \wedge p \rightarrow q$  is state valid<sup>2</sup>. The conclusion is that the implication  $p \rightarrow q$  is  $P$ -state valid.

In most cases we apply rule sv-PSV to a verification condition  $\{\varphi\} \tau \{\psi\}$ . Taking  $p$  to be  $\rho_\tau \wedge \varphi$  and  $q$  to be  $\psi'$ , we obtain the derivation

<sup>2</sup> For the rule to be sound, it is sufficient that the implication  $\chi \wedge p \rightarrow q$  be  $P$ -state valid. However, in practice state validity is most often used.

$$\frac{P \models \square X, \quad \models \{X \wedge \varphi\} \tau \{\psi\}}{P \models \{\varphi\} \tau \{\psi\}}$$

Assume that we have established the  $P$ -invariance of assertion  $X$  in a previous proof, and we intend to use rule INV-B to establish the  $P$ -invariance of assertion  $\varphi$ . According to the preceding derivation, it is sufficient to establish the state validity of the two implications

- B1.  $\Theta \rightarrow \varphi$
- B2.  $\{X \wedge \varphi\} \tau \{\varphi\}$ .

In the case that these implications are state valid, we say that assertion  $\varphi$  is *inductive relative* to  $X$ .

This establishes a pattern we often follow in proving invariants. First, we establish some verification conditions directly from state validities. With those we establish some  $P$ -invariants. Then, we use these  $P$ -invariants to establish the  $P$ -validity of more complex verification conditions. We refer to this style of proof as *incremental*.

### Example (mutual exclusion by semaphores)

We illustrate the process of incremental proof on program MUX-SEM (Fig. 1.6) presented in the example above. Assume we have already established the  $P$ -invariance of

$$\varphi_1: \quad at\_l_{3,4} + at\_m_{3,4} + y = 1.$$

We use rule INV-B in conjunction with sv-PSV (in which we take  $X$  to be  $\varphi_1$ ) to derive the property of mutual exclusion  $\square \varphi_2$ , where

$$\varphi_2: \quad \neg(at\_l_3 \wedge at\_m_3).$$

We proceed to establish the premises of rule INV-B for  $\varphi_2$ .

- *Establishing  $\Theta \rightarrow \varphi_2$ :*

Premise B1 requires:

$$\underbrace{\pi = \{l_0, m_0\} \wedge y = 1}_{\Theta} \rightarrow \underbrace{\neg(at\_l_3 \wedge at\_m_3)}_{\varphi_2}$$

which is obviously valid.

- *Establishing  $\{\varphi_1 \wedge \varphi_2\} \tau \{\varphi_2\}$ :*

It remains to establish, under premise B2, the verification condition for all diligent transitions  $\tau$  in the program, where we may uniformly add  $\varphi_1$  to the left-hand sides of all implications. This calls for showing, for each diligent transition  $\tau$ , the validity of the implication

$$\rho_T \wedge \underbrace{at\text{-}\ell_{3,4} + at\text{-}m_{3,4} + y = 1}_{\chi = \varphi_1} \wedge \underbrace{\neg(at\text{-}\ell_3 \wedge at\text{-}m_3)}_{\varphi_2} \rightarrow \underbrace{\neg(at'\text{-}\ell_3 \wedge at'\text{-}m_3)}_{\varphi'_2}.$$

We consider first transitions of  $P_1$ . For all  $P_1$ -transitions except for  $\ell_2$ ,  $\rho_T$  implies  $at'\text{-}\ell_3 = F$  and  $\varphi'_2$  is obviously true. It remains to check the implication for  $\ell_2$ .

- Establishing  $\{\varphi_1 \wedge \varphi_2\} \ell_2 \{\varphi_2\}$ :

To establish the implication for  $\ell_2$ , we observe that  $\rho_{\ell_2}$  implies  $y > 0$ . Thus, it suffices to verify

$$y > 0 \wedge at\text{-}m_{3,4} + y = 1 \rightarrow \neg at\text{-}m_3.$$

But from the left-hand side of the implication we infer

$$at\text{-}m_{3,4} = 1 - y < 1,$$

which can only hold if  $at\text{-}m_{3,4} = 0 = F$ , and hence  $\neg at\text{-}m_3$ .

The transitions of  $P_2$  can be similarly checked.

By rule INV-B, we therefore conclude the desired property

$$\square \neg(at\text{-}\ell_3 \wedge at\text{-}m_3). \blacksquare$$

## Incremental Proofs vs. Assertion Strengthening

Let  $\varphi_2$  be an assertion that is invariant but not inductive. The preceding discussion identified two ways to proceed.

One possible strategy strengthens  $\varphi_2$  by considering the conjunction  $\varphi_1 \wedge \varphi_2$ , and applies rule INV to the stronger assertion. The list of state formulas that should be proven to be  $P$ -state valid is

- S1.  $\Theta \rightarrow (\varphi_1 \wedge \varphi_2)$
- S2.  $\{\varphi_1 \wedge \varphi_2\} T \{\varphi_1 \wedge \varphi_2\}$ .

These formulas correspond to premises I2 and I3 of rule INV. Premise I1 is not represented, since the implication  $\varphi_1 \wedge \varphi_2 \rightarrow \varphi_2$  is trivially state valid.

An alternate strategy is to identify an inductive assertion  $\varphi_1$ , such that  $\varphi_2$  is inductive relative to  $\varphi_1$ , and then use rule INV-B. The list of state formulas that have to be proven state valid is given by

- J1.  $\Theta \rightarrow \varphi_1$
- J2.  $\{\varphi_1\} T \{\varphi_1\}$

$$J3. \quad \Theta \rightarrow \varphi_2$$

$$J4. \quad \{\varphi_1 \wedge \varphi_2\} T \{\varphi_2\},$$

where J1, J2 are needed to verify  $\Box \varphi_1$  by rule INV-B and J3, J4 are needed for the proof of  $\Box \varphi_2$ .

An important observation is that the state validity of J1–J4 implies the state validity of S1, S2. The less trivial part of this observation is to show that S2 is implied by J2 and J4. Indeed, for every  $\tau \in T$ ,

$$\rho_\tau \wedge \varphi_1 \rightarrow \varphi'_1 \quad \text{and} \quad \rho_\tau \wedge \varphi_1 \wedge \varphi_2 \rightarrow \varphi'_2$$

obviously imply the condition

$$\rho_\tau \wedge \varphi_1 \wedge \varphi_2 \rightarrow \varphi'_1 \wedge \varphi'_2.$$

This observation shows that every incremental proof using assertions  $\varphi_1$  and  $\varphi_2$  can be uniformly converted to a proof by rule INV using the strengthened assertion  $\varphi_1 \wedge \varphi_2$ . This establishes INV as a rule which is at least as general as the method of incremental proofs.

There are cases in which the conjunction  $\varphi_1 \wedge \varphi_2$  is inductive, but it is not the case that  $\varphi_1$  is inductive and  $\varphi_2$  is inductive relative to  $\varphi_1$ . Consider, for example, program INCREMENT presented in Fig. 1.9.

```
local x: integer where x = 1
```

```
l0: loop forever do [l1: x := x + 1]
```

Fig. 1.9. Program INCREMENT.

Assertion  $\varphi_2$ :  $at-l_1 \rightarrow x > 0$  is an invariant of this program but is not inductive. The strengthening strategy can be applied by conjoining the assertion  $\varphi_1$ :  $at-l_0 \rightarrow x > 0$  to  $\varphi_2$ , and obtaining the inductive assertion  $\varphi_1 \wedge \varphi_2$  which can be used in rule INV to establish the invariance of  $\varphi_2$ . Unfortunately,  $\varphi_1$  by itself is not inductive. Consequently, we cannot use incremental proofs with  $\varphi_1$  and  $\varphi_2$  to prove the invariance of  $\varphi_2$  without introducing additional (strengthening) assertions.

In spite of this example indicating that incremental proof is sometimes a weaker method than rule INV, we strongly recommend its use whenever applicable. Its main advantage is that of *modularity*. That is, we deal separately with  $\varphi_1$  and then prove verification conditions of  $\varphi_2$  with the help of  $\varphi_1$ . In contrast, rule INV for the same case immediately reasons about the full conjunction  $\varphi_1 \wedge \varphi_2$ .

### Combining the Strategies

It is possible to combine the use of a stronger auxiliary assertion and previously established invariants and obtain the advantages of both methods. This is formulated in rule INC-INV (Fig. 1.10).

For assertions  $p, \varphi, p_1, \dots, p_k$ ,

$$\text{I0. } \square p_1, \dots, \square p_k$$

$$\text{I1. } \left( \bigwedge_{i=1}^k p_i \right) \wedge \varphi \rightarrow p$$

$$\text{I2. } \Theta \rightarrow \varphi$$

$$\text{I3. } \left\{ \left( \bigwedge_{i=1}^k p_i \right) \wedge \varphi \right\} T \{\varphi\}$$

---


$$\square p$$

Fig. 1.10. Rule INC-INV (incremental invariance).

The rule assumes that assertions  $p_1, \dots, p_k$  have been previously shown to be  $P$ -invariant. It follows that each  $P$ -accessible state satisfies the conjunction  $\chi: \bigwedge_{i=1}^k p_i$ , and we append it to the left-hand side of the implications in premises I1 and I3. In principle, it is also possible to add  $\bigwedge_{i=1}^k p_i$  to the left-hand side of I2, but no  $p_i$  can contain additional initial information that is not implied by  $\Theta$ .

### Detecting Trivial Verification Conditions

In theory, when we establish a verification condition of the form

$$\{\varphi\} T \{\varphi\},$$

we have to consider all transitions in  $P$ . In practice, it is often possible to identify some verification conditions as trivially holding and to concentrate on the more difficult ones. We now discuss some clues for recognizing trivial cases. Note that we already mentioned that there is no need to check the verification condition for the idling transition  $\tau_I$ .

### Unmodified Assertions

The first obvious case of trivial verification conditions is related to transitions that do not modify any variable on which  $\varphi$  depends. In this case, obviously, if

$\varphi$  holds before the transition it will also hold after the transition. An example of such a case is the transition  $\ell_3$  in program MUX-SEM (Fig. 1.6), which preserves the assertion  $y \geq 0$  since it does not modify  $y$ .

A more general case is an assertion  $\varphi$  that depends on a variable  $y$  only via an expression  $\mathcal{E}(y)$  and a transition  $\tau$  that does modify  $y$  but does so in a way that preserves the value of  $\mathcal{E}(y)$ . A characteristic case is when  $\varphi$  depends on the program locations only via expressions of the form  $at\_L_1, at\_L_2, \dots$ , etc., where each  $L_i$  is a set of locations (a region) in the program. Then, any transition which neither enters nor exits any of these regions preserves the values of these location expressions. For example, in program MUX-SEM,  $at\_\ell_{3,4}$  is not changed by any transition  $m_i$ , nor is it changed by  $\ell_3$ , which exits  $\ell_3$  but enters  $\ell_4$  at the same time.

Thus, if  $\varphi$  depends on data variables and location expressions whose values are not modified by transition  $\tau$ , then  $\tau$  preserves  $\varphi$ . As an example, in program MUX-SEM, transition  $\ell_3$  preserves the assertion  $at\_\ell_{3,4} + at\_m_{3,4} + y = 1$ , since it does not enter or exit the location sets  $\ell_{3,4}$  or  $m_{3,4}$  and does not modify  $y$ .

A similar argument can be applied to process-specific assertions. Let  $P_i$  be a process such that  $\varphi$  depends only on locations within  $P_i$  and on data variables which are only modified by  $P_i$ . Then the only transitions to be checked when establishing the verification conditions for  $\varphi$  are transitions in  $P_i$ , including entries and exits from  $P_i$  if there are any. We can safely ignore transitions lying completely outside  $P_i$ .

We say that a transition  $\tau$  *interferes* with an assertion  $\varphi$  if it modifies a data variable or a location expression on which  $\varphi$  depends.

We may summarize this by:

### Suggestion 1

Ignore transitions that do not interfere with  $\varphi$ .

### Implications

Another frequent case is presented by assertions that have the form of implications  $\varphi$ :  $p \rightarrow q$  leading to a verification condition of the form

$$\rho_\tau \wedge \underbrace{p \rightarrow q}_{\varphi} \rightarrow \underbrace{p' \rightarrow q'}_{\varphi'}$$

In general, we should only be concerned about transitions that may possibly falsify  $\varphi$ , i.e., cause it to change from T to F. We refer to such transitions as *potentially falsifying* transitions.

For the case of an assertion that is an implication  $p \rightarrow q$ , the only potentially falsifying transitions are those that may *validate*  $p$ , i.e., change  $p$  from F to T, or

those that may *falsify*  $q$ , i.e., change  $q$  from  $\top$  to  $\perp$ . It is easy to see that a transition which neither validates  $p$  nor falsifies  $q$  cannot falsify the full implication  $p \rightarrow q$ .

We may summarize this discussion by the following recommendation:

### Suggestion 2

For assertions of the form  $p \rightarrow q$ , consider only transitions that may potentially validate  $p$  or falsify  $q$ .

We can identify the type of transitions that may validate  $p$  or falsify  $q$  for some special cases.

- *A subformula of the form at- $L$ .*

Such a subformula can be validated only by transitions of the form  $\ell \rightsquigarrow \tilde{\ell}$ , i.e., transitions that move from  $\ell$  to  $\tilde{\ell}$ , where  $\ell \notin L$  and  $\tilde{\ell} \in L$ . That is, transitions that enter  $L$ .

It can be falsified only by transitions  $\ell \rightsquigarrow \tilde{\ell}$  that exit  $L$ , i.e., such that  $\ell \in L$  and  $\tilde{\ell} \notin L$ .

It is not affected at all by transitions  $\ell \rightsquigarrow \tilde{\ell}$  such that either  $\ell, \tilde{\ell} \in L$  or  $\ell, \tilde{\ell} \notin L$ .

- *A subformula of the form at- $L_1 \wedge$  at- $L_2$ , with  $L_1$  and  $L_2$  belonging to parallel processes.*

Excluding from our discussion synchronous communication transitions that can enter both  $L_1$  and  $L_2$  at the same time, such a subformula can be validated only by transitions that either enter  $L_1$  or enter  $L_2$ . Similarly, the conjunction  $at\_L_1 \wedge at\_L_2$  can be falsified only by transitions that either exit  $L_1$  or exit  $L_2$ .

For example, if the full assertion is of the form

$$\varphi: \quad at\_L_1 \wedge at\_L_2 \rightarrow Q$$

then, for a transition  $\tau$  that enters  $L_1$ ,  $at\_L_1 \wedge at\_L_2$  can change from false to true only if  $at\_L_2$  was already true before. Consequently, to verify  $\{\varphi\} \tau \{\varphi\}$  it suffices to prove

$$\{at\_L_2\} \tau \{Q\},$$

i.e.,  $\rho_\tau \wedge at\_L_2 \rightarrow Q'$ .

We refer to this case of a verification condition as:

$$\tau \text{ while } at\_L_2.$$

**Example** (mutual exclusion by semaphores)

As an example, reconsider the verification of the invariant

$$\varphi: \quad \neg(at\_l_3 \wedge at\_m_3)$$

for the semaphore program MUX-SEM (Fig. 1.6), which can be rewritten as

$$\underbrace{at_{-\ell_3} \wedge at_{-m_3}}_p \rightarrow \underbrace{F}_q.$$

Here the location sets are  $L_1 = \{\ell_3\}$  and  $L_2 = \{m_3\}$ , respectively. Assume that we have already established the invariant

$$at_{-\ell_3,4} + at_{-m_3,4} + y = 1.$$

According to suggestion 2, we should only consider the following two entry transitions that may validate  $at_{-\ell_3} \wedge at_{-m_3}$ :

- $\ell_2 \rightsquigarrow \ell_3$  while  $at_{-m_3}$ .

This case requires the verification of  $\{at_{-m_3}\} \tau_{\ell_2} \{F\}$ , that is,

$$\rho_{\ell_2} \wedge at_{-m_3} \rightarrow F.$$

Since we are allowed to add to the antecedent any assertion whose invariance has been previously established (rule INC-INV), it suffices to prove:

$$\rho_{\ell_2} \wedge at_{-\ell_3,4} + at_{-m_3,4} + y = 1 \wedge at_{-m_3} \rightarrow F,$$

which is obviously true, since  $at_{-\ell_3,4} + at_{-m_3,4} + y = 1$  and  $at_{-m_3} = 1$  imply  $y \leq 0$ , while  $\rho_{\ell_2}$  implies  $y > 0$ .

- $m_2 \rightsquigarrow m_3$  while  $at_{-\ell_3}$ .

This case can be treated similarly. ■

### 1.3 Finding Inductive Assertions: The Bottom-Up Approach

The strengthening strategy, formulated in the preceding section, suggests that, in order to prove that an assertion  $p$  is  $P$ -invariant, we find a stronger assertion  $\varphi$  (that is,  $\varphi \rightarrow p$ ), which is inductive. Indeed, as will be shown in Section 2.5, this single strategy is adequate for proving all the invariance properties of any given program. This makes the task of finding an inductive assertion  $\varphi$  that strengthens a given  $p$  one of the most important tasks in verification.

In this and the following section, we outline some possible techniques for identifying and constructing inductive assertions. The methods we consider for the construction of inductive assertions can be partitioned into bottom-up and top-down methods. In the *bottom-up* approach we only consider the given program and attempt to deduce from its structure assertions that are inductive. In the *top-down* approach, we are also guided by the given goal assertion  $p$ , whose invariance we wish to establish. In this section we consider bottom-up approaches.

## Transition-Validated Assertions

The simplest type of bottom-up identifiable inductive assertions are those which are guaranteed to hold after execution of each transition that interferes with them, without any assumption about the state before the execution. These are assertions  $\varphi$  such that, for every transition  $\tau$  interfering with  $\varphi$ ,

$$\rho_\tau \rightarrow \varphi'.$$

Inductiveness also requires that the initial condition  $\Theta$  implies  $\varphi$ , that is,

$$\Theta \rightarrow \varphi,$$

and that should also be checked. We say that these assertions are transition validated, since every interfering transition validates them, i.e., causes them to become true.

Examples of such inductive assertions are provided by locations  $\ell$  in the program that are reachable only as a result of a test  $\kappa$ . In such a case we know that when we first enter this location the test is valid. If, in addition, no other transition can modify the variables on which the test  $\kappa$  depends, as long as control stays in this location, then an inductive assertion is

$$at\_ \ell \rightarrow \kappa.$$

For instance, if the program contains the *while* statement

$$\ell_1: [\text{while } c \text{ do } S]; \quad \ell_2:$$

and no statement parallel to  $\ell_1$  can modify the variables on which  $c$  depends, then we can easily deduce that

$$at\_ \ell_2 \rightarrow \neg c$$

is an inductive assertion.

Similar examples of such inductive assertions arise if a location  $\ell_2$  is reachable only by a transition that sets a variable  $y$  to a constant  $a$

$$\ell_1: y := a; \quad \ell_2:$$

If  $y$  is not modifiable by a parallel transition as long as we stay at  $\ell_2$ , we may conclude the inductiveness of

$$at\_ \ell_2 \rightarrow y = a.$$

## Single-Variable Assertions

Simple inductive assertions that refer to a single data variable are sometimes suggested by observing that this variable is modified only in a restricted and predictable way. A typical case is the semaphore variable discussed earlier.

If  $y$  is a variable whose initial value is nonnegative, and the only writing references to it are via **request**  $y$  and **release**  $y$ , then  $y \geq 0$  is an inductive assertion of the program.

In other examples, we may establish that an arbitrary integer variable is nonnegative, is even, etc.

## Multi-Variable Assertions

Inductive assertions involving more than one variable may be suggested by observing that members of a particular set of variables are always modified together.

### Example (integer square root)

Consider program **SQUARE-ROOT**, presented in Fig. 1.11, for the computation of the integer square root of a nonnegative integer  $x$ .

```

in   x   : integer where x ≥ 0
local u, w: integer where u = 1, w = 1
out  z   : integer where z = 0

ℓ₀: while w ≤ x do
    ℓ₁: (z, u, w) := (z + 1, u + 2, w + u + 2)
    ℓ₂:

```

Fig. 1.11. Program **SQUARE-ROOT** (integer square root).

This program employs a multiple assignment that assigns values to  $z$ ,  $u$ , and  $w$  in one step. In addition to the idling transition, the program has two transitions  $\ell_0$  and  $\ell_1$ , whose transition relations are given by

$$\rho_0: \underbrace{\text{move}(\ell_0, \ell_1) \wedge w \leq x}_{\rho_0^T} \vee \underbrace{\text{move}(\ell_0, \ell_2) \wedge w > x}_{\rho_0^F}$$

$$\rho_1: \text{move}(\ell_1, \ell_0) \wedge z' = z + 1 \wedge u' = u + 2 \wedge w' = w + u + 2.$$

Partial correctness of this program can be specified by the assertion

$$\psi: \text{at-}\ell_2 \rightarrow z^2 \leq x < (z + 1)^2,$$

which states that on termination,  $z$  is the largest integer whose square does not exceed  $x$ .

The intuition behind this program is that the variable  $z$  ranges over the sequence of the natural numbers  $0, 1, \dots, n$  while  $u$  ranges over the sequence  $1, 3, \dots, 2n+1$ , and  $w$  maintains the sum  $1+3+\dots+(2n+1) = (n+1)^2 = (z+1)^2$ . The loop terminates the first time that  $w > x$ , i.e.,  $x < (z+1)^2$ . Since it did not terminate in the previous iteration it follows that  $z^2 \leq x$ . We thus conclude that on termination  $z^2 \leq x < (z+1)^2$ .

We wish to show that the invariant  $\psi$  can be established by the systematic approaches discussed in this section. This specification consists of two parts, which assure that on termination  $z^2 \leq x$  and  $x < (z+1)^2$ . We first establish the invariance of the second part, that is,

$$\psi_2: \text{at-}\ell_2 \rightarrow x < (z+1)^2.$$

Examining the modifications of the three variables  $u$ ,  $w$ , and  $z$ , we observe that whenever  $z$  is incremented by 1,  $u$  is incremented by 2. This suggests that the difference  $u - 2 \cdot z$  remains invariant. We also find that the initial values of  $u$  and  $z$  are 1 and 0. Consequently we have the following assertion:

$$\varphi_1: u = 2 \cdot z + 1.$$

It is not difficult to check that this is indeed an inductive assertion of the above program.

Another observation is that  $u$  and  $w$  are also modified at the same time. However, the relation between their respective modifications is more complicated. Let us denote by  $z_n$ ,  $u_n$ ,  $w_n$  the values assigned to  $z$ ,  $u$ ,  $w$ , respectively, on their  $n$ th modification. We will denote by  $z_0$ ,  $u_0$ ,  $w_0$  their initial values. It is clear that the following recurrence equations hold:

$$\begin{cases} z_0 = 0 \\ z_n = z_{n-1} + 1 \quad \text{for } n > 0 \end{cases}$$

$$\begin{cases} u_0 = 1 \\ u_n = u_{n-1} + 2 \quad \text{for } n > 0 \end{cases}$$

$$\begin{cases} w_0 = 1 \\ w_n = w_{n-1} + u_{n-1} + 2 \quad \text{for } n > 0. \end{cases}$$

For completeness, we rederive the invariant  $\varphi_1$ , this time using the recurrence equations.

- *Deriving  $\varphi_1$ :*

The recurrence equations for  $z_n$  and  $u_n$  can be solved first, since  $z_n$  and  $u_n$  do not depend on the  $w_i$ 's. The solution is

$$\begin{aligned} z_n &= n && \text{for } n \geq 0 \\ u_n &= 2 \cdot n + 1 && \text{for } n \geq 0; \end{aligned}$$

and therefore, eliminating  $n$  between the two equations, and omitting the  $n$ -subscript, we obtain

$$\varphi_1: u = 2 \cdot z + 1.$$

- *Deriving a second invariant  $\varphi_2$ :*

Substituting the solution for  $u_n$  in the equation for  $w_n$  we obtain

$$\begin{cases} w_0 = 1 \\ w_n = w_{n-1} + (2 \cdot (n-1) + 1) + 2 \\ \quad = w_{n-1} + (2n+1) \end{cases} \quad \text{for } n > 0,$$

which can be solved to get

$$w_n = \sum_{k=0}^n (2k+1) = (n+1)^2 \quad \text{for } n \geq 0.$$

We observe that  $n$  can be eliminated by using  $z_n$  instead:

$$w_n = (z_n + 1)^2 \quad \text{for } n \geq 0,$$

leading to

$$\varphi_2: w = (z+1)^2$$

as a candidate for an invariant. It can be checked that  $\varphi_2$  is an inductive assertion (relative to  $\varphi_1$ ).

In fact, any assertion that is derived by the solution of such recurrence equations is guaranteed to be inductive, and there is no need to recheck its inductiveness.

- *Establishing  $\psi_2$ :*

We can use the invariance of  $\varphi_2$  to show the inductiveness of

$$\psi_2: at\_\ell_2 \rightarrow x < (z+1)^2.$$

The only potentially falsifying transition for  $\psi_2$  is  $\ell_0$  in the mode  $\ell_0^F$  which enters  $\ell_2$ . The verification condition for  $\ell_0^F$  and  $\psi_2$  is

$$\underbrace{\dots \wedge w > x \wedge z' = z \wedge \dots}_{\rho_{\ell_0^F}} \wedge \underbrace{\dots}_{\psi_2} \rightarrow \underbrace{\dots \rightarrow x < (z'+1)^2}_{\psi'_2}.$$

Obviously,  $\varphi_2: w = (z+1)^2$  and  $w > x \wedge z' = z$  imply  $x < (z'+1)^2$ .

This yields the invariance of only half of the necessary requirement  $\psi$ . To prove the other half, we will use the top-down approach discussed in the next section.



## Control Invariants

There are several useful bottom-up invariants which constrain the possible values of the control variable  $\pi$ . Here we present some of these control invariants. Each of them can be proven  $P$ -invariant by rule INV, but we omit the proofs.

- Control invariant **CONFLICT**

For labels  $\ell_i, \ell_j$  which are in conflict (see page 16),

$$\neg(at_{-\ell_i} \wedge at_{-\ell_j}).$$

This invariant states that control cannot reside at two conflicting locations at the same time.

For example, we can use invariant CONFLICT to infer

$$at_{-\ell_{0..2}} \rightarrow \neg at_{-\ell_{3,4}}$$

for program MUX-SEM of Fig. 1.6.

- Control invariant **SOMEWHERE**

For each  $P_i$ , a top-level process of the program, let  $\mathcal{L}_i$  denote the set of all locations within  $P_i$ . The following invariant states that control is always somewhere within process  $P_i$ :

$$\bigvee_{\ell \in \mathcal{L}_i} at_{-\ell}$$

For example, we can use invariant SOMEWHERE to infer

$$|\pi_\ell| > 0 \quad \text{and} \quad |\pi_m| > 0$$

for program MUX-SEM of Fig. 1.6.

- Control invariant **EQUAL**

For labels  $\ell, m$ , where  $\ell \sim_L m$ ,

$$at_{-\ell} \leftrightarrow at_{-m}.$$

This invariant states that if labels  $\ell$  and  $m$  are equivalent, i.e.,  $[\ell] = [m]$ , then  $at_{-\ell}$  is equivalent to  $at_{-m}$  on all  $P$ -accessible states.

Thus, if

$$\left[ \begin{array}{l} \ell_1: x := 1 \\ \text{or} \\ \ell_2: y := 2 \end{array} \right]$$

is a substatement of program  $P$ , then

$$at_{-\ell_1} \leftrightarrow at_{-\ell_2}$$

is an instance of control invariant EQUAL for  $P$ .

- Control invariant PARALLEL

For  $S_1 \parallel S_2$  a substatement of program  $P$

$$\text{in\_}S_1 \leftrightarrow \text{in\_}S_2.$$

Invariant PARALLEL represents the fact that a cooperation statement starts its execution by entering all its children at the same time, and can terminate only when all children have terminated.

Consider, for example, a program  $P$  containing the substatement

$$\left[ \begin{array}{l} \ell_1: x := 1 \\ \ell_2: \end{array} \right] \quad \parallel \quad \left[ \begin{array}{l} m_1: y := 1 \\ m_2: \end{array} \right].$$

An instance of invariant PARALLEL for  $P$  is

$$\text{at\_}\ell_1 \vee \text{at\_}\ell_2 \leftrightarrow \text{at\_}m_1 \vee \text{at\_}m_2.$$

In Problem 1.3, we ask the reader to establish the  $P$ -invariance of control invariants CONFLICT, SOMEWHERE, EQUAL, and PARALLEL.

From now on, whenever we conduct a proof, using rules INV or INV-B, we assume that all the control invariants have already been proven for the considered program. Technically, this means that we can add any instance of these invariants to the antecedent of any of the premises or, equivalently, use the control invariants to simplify the premises and verification conditions. Also, whenever we say that an assertion is inductive, we implicitly include the control invariants as members of the set of pre-established invariants.

### Example (mutual exclusion by semaphores)

Reconsider program MUX-SEM of Fig. 1.6. In a previous analysis of this program (see page 94), we established mutual exclusion expressed by the invariance of the assertion  $p: \neg(\text{at\_}\ell_3 \wedge \text{at\_}m_3)$ . The proof was based on strengthening  $p$  into the inductive assertion

$$\varphi: |\pi_\ell| = |\pi_m| = 1 \wedge y \geq 0 \wedge \text{at\_}\ell_{3,4} + \text{at\_}m_{3,4} + y = 1.$$

Now that we routinely assume that all control invariants are pre-established and available, we can repeat the proof with the simpler assertion

$$\widehat{\varphi}: y \geq 0 \wedge \text{at\_}\ell_{3,4} + \text{at\_}m_{3,4} + y = 1.$$

This is because the conjunct  $|\pi_\ell| = |\pi_m| = 1$  is implied by control invariants CONFLICT and SOMEWHERE of program MUX-SEM. Note that  $\widehat{\varphi}$  is inductive relative to the control invariants. ■

## 1.4 Finding Inductive Assertions: The Top-Down Approach

The bottom-up approach, discussed above, is based on making local observations uninfluenced by the goal we wish to prove. The top-down approach is goal-directed in that it considers the property to be proved and strengthens some of its parts to produce an inductive assertion.

We can offer the following comparison between the two approaches:

The bottom-up approach is

- algorithmic,
- solely based on the program's text, and
- guaranteed to produce inductive assertions. There is no need to check that the resulting assertion is inductive.

The top-down approach is

- heuristic,
- based on the goal assertion as well as on the program's text, and
- not guaranteed to produce an inductive assertion. The inductiveness of the resulting assertion still has to be independently checked.

Assume that we are given a goal assertion  $\psi$  whose invariance we wish to establish. As a first step, we check whether  $\psi$  is inductive relative to  $\chi$ , the conjunction of all the assertions that have so far been proven to be invariant. If  $\psi$  is found to be inductive relative to  $\chi$ , we are done.

The case requiring further attention is when  $\psi$  is not inductive relative to  $\chi$ . The main strategy to be followed is to strengthen  $\psi$  into an inductive assertion  $\varphi$ . In this section, we present several heuristics and techniques for the strengthening process.

### Strengthening Heuristics

To introduce two strengthening heuristics, consider program SUM presented in Fig. 1.12.

This program consists of a single *while* statement that accumulates in variable *sum* the value of  $\sum_{r=1}^N A[r]$ . Consequently, partial correctness of this program is stated by the invariance of the assertion

$$\psi: \text{at\_}\ell_2 \rightarrow \text{sum} = \sum_{r=1}^N A[r].$$

```

in   N : integer where  $N \geq 0$ 
      A : array [1..N] of integer
local i : integer where  $i = 0$ 
out sum: integer where  $sum = 0$ 

```

$\ell_0$ : **while**  $i \neq N$  **do**

$\ell_1$ :  $(i, sum) := (i + 1, sum + A[i + 1])$

$\ell_2$ :

Fig. 1.12. Program SUM.

This assertion is not inductive by itself.

The first heuristic we apply for strengthening  $\psi$  can be described as follows:

#### Heuristic (generalization)

In an assertion, replace references to constants (or input variables) by references to variables whose values are known to equal those constants in the relevant location range. Also add a clause that explicitly states this equality.

We may follow this heuristic and replace the reference to  $N$  in  $\psi$  by the variable  $i$  that is known to equal  $N$  when control reaches  $\ell_2$ . We also add the conjunct  $i = N$  that explicitly states this fact. This leads to the strengthened assertion

$$\varphi: \quad at\_{\ell_2} \rightarrow \sum_{r=1}^i A[r] \wedge i = N.$$

The generalization heuristic is much more than a symbol manipulation transformation that can be applied arbitrarily to any reference to a constant. A successful application of this heuristic must be based on a good understanding of the complete task the program is expected to accomplish, and how intermediate situations in the execution of the program correspond to partial fulfillment of this task. In the example of the considered program, the complete task corresponds to the accumulation of the sum for  $A[1], \dots, A[N]$ . This task is fully accomplished only on termination, as is stated by  $\psi$ . Intermediate states in the computation of the program will find the control somewhere between  $\ell_0$  and  $\ell_1$ , and can use the variable  $i$  as a measure of progress in accomplishing the summation task. Thus, at intermediate states,  $sum$  always contains the accumulated sum of  $A[1], \dots, A[i]$ . This is precisely stated by the proposed invariant clause  $sum = \sum_{r=1}^i A[r]$ .

It is this understanding of the way the program accomplishes its task that

guided us to replace the reference to  $N$ , rather than the reference to the other constant 1. Assume that instead of program SUM we were considering a similar summation program, call it SUM-DOWN, in which  $i$  is initially set to  $N$  and then decreased until it reaches 0. The original specification  $\psi$  is also an invariant of program SUM-DOWN. However, the correct generalization in this case is given by

$$\tilde{\varphi}: \text{at-}\ell_2 \rightarrow \text{sum} = \sum_{r=i+1}^N A[r] \wedge i = 0.$$

Let us continue with the proof of  $\varphi$ . The next step is to split  $\varphi$  into a conjunction of the two assertions

$$\varphi_1: \text{at-}\ell_2 \rightarrow \text{sum} = \sum_{r=1}^i A[r]$$

$$\varphi_2: \text{at-}\ell_2 \rightarrow i = N,$$

and try to prove their invariance independently.

It is not difficult to ascertain that  $\varphi_2$  is already inductive. The only relevant transition is the F-mode of  $\ell_0$ , and its transition relation implies  $i = N$ .

The assertion  $\varphi_1$ , on the other hand, is not yet inductive and therefore requires additional strengthening. We observe that the assertion  $\varphi_1$  has the form  $K \rightarrow \delta$ , where  $K$  is a *control assertion*, i.e., an assertion referring only to the control variable  $\pi$ ; and  $\delta$  is a *data assertion*, i.e., an assertion referring only to the data variables. To strengthen the assertion  $\varphi_1$ , we apply a second heuristic that can be formulated as follows:

### Heuristic (range extension)

If the candidate assertion has the form  $\varphi: K \rightarrow \delta$ , where  $K$  is a control assertion and  $\delta$  is a data assertion, strengthen it into the assertion  $\hat{\varphi}: \hat{K} \rightarrow \delta$ , where  $K \rightarrow \hat{K}$ . As the simplest candidate, take  $\hat{K} = \top$ , which leads to  $\hat{\varphi}: \delta$ .

The assertion  $\varphi: K \rightarrow \delta$  states that whenever the control variable satisfies  $K$ , the data variables should satisfy  $\delta$ . The implication  $K \rightarrow \hat{K}$  means that the set of states satisfying  $\hat{K}$  is an extension (superset) of the set of states satisfying  $K$ . Thus,  $\hat{\varphi}$  states that  $\delta$  holds on additional states. Taking  $\hat{K}$  to be  $\top$  means that  $\delta$  should hold on all accessible states.

Applying this extreme case of the range-extension heuristic to  $\varphi_1$  we obtain the assertion

$$\hat{\varphi}_1: \text{sum} = \sum_{r=1}^i A[r]$$

which can be shown to be inductive.

## Assertion Propagation

A very powerful and useful strengthening heuristic is based on the identification of an unprovable verification condition. Assume that  $\varphi$  is an invariant assertion that cannot be shown to be inductive relative to the previously proven invariant  $\chi$ . This means that there exists some transition  $\tau$  such that the verification condition

$$\{\chi \wedge \varphi\} \tau \{\varphi\}$$

is not state valid.

We define the *precondition* of  $\varphi$  with respect to  $\tau$ , denoted  $\text{pre}(\tau, \varphi)$ , to be the assertion

$$\text{pre}(\tau, \varphi): \forall V': \rho_\tau \rightarrow \varphi'.$$

This formula characterizes all the states that are  $\tau$ -predecessors of a  $\varphi$ -state. That is, a state  $s$  satisfies  $\text{pre}(\tau, \varphi)$  iff all  $\tau$ -successors of  $s$  satisfy  $\varphi$ .

Consider, for example, the case that  $V: \{x\}$ ,  $\rho_\tau: x' = x + 2$ , and  $\varphi: x = 2$ . The precondition  $\text{pre}(\tau, \varphi)$  is given by

$$\forall x': \underbrace{x' = x + 2}_{\rho_\tau} \rightarrow \underbrace{x' = 2}_{\varphi'}$$

which can be simplified to

$$x = 0.$$

Indeed,  $x = 0$  at a state  $s$  iff all  $\tau$ -successors of  $s$  satisfy  $x = 2$ .

Since all transition relations occurring in programs have the form

$$\rho_\tau: C_\tau \wedge \overline{V}' = \overline{E},$$

where  $C_\tau$  is the enabling condition of  $\tau$ , the generated precondition

$$\text{pre}(\tau, \varphi): \forall V': C_\tau \wedge \overline{V}' = \overline{E} \rightarrow \varphi'$$

can always be simplified to

$$C_\tau \rightarrow \varphi'[\overline{V}' \mapsto \overline{E}].$$

Assuming that  $\overline{V} = (v_1, \dots, v_m)$  and  $\overline{E} = (e_1, \dots, e_m)$ ,  $\varphi'[\overline{V}' \mapsto \overline{E}]$  stands for the assertion obtained from  $\varphi'$  by replacing every occurrence of  $v'_i$  by the expression  $e_i$ , for each  $i = 1, \dots, m$ . In many cases, we can perform simplifications on  $C_\tau \wedge \overline{V}' = \overline{E} \rightarrow \varphi'$  prior to the elimination of quantifiers by substitution. These simplifications are particularly recommended for the control predicates.

**Example** Consider, for example, program ADD-TWO presented in Fig. 1.2 and the assertion

$$\varphi: \text{at\_}\ell_1 \rightarrow x = 2.$$

The precondition of  $\varphi$  with respect to transition  $\ell_0$  is given by

$$\forall \pi', x': \underbrace{\text{move}(\ell_0, \ell_1) \wedge x' = x + 2}_{\rho_{\ell_0}} \rightarrow \underbrace{\text{at}'_{-\ell_1} \rightarrow x' = 2}_{\varphi'}.$$

Since

$$\text{move}(\ell_0, \ell_1): \text{at}_{-\ell_0} \wedge \pi' = \pi - \{\ell_0\} \cup \{\ell_1\}$$

implies  $\text{at}'_{-\ell_1} = \top$ , this formula can be simplified to

$$\forall \pi', x': \text{at}_{-\ell_0} \wedge \pi' = \pi - \{\ell_0\} \cup \{\ell_1\} \wedge x' = x + 2 \rightarrow x + 2 = 2$$

which is equivalent to

$$\text{pre}(\ell_0, \varphi): \text{at}_{-\ell_0} \rightarrow x = 0. \blacksquare$$

**Claim** If  $\varphi$  is  $P$ -invariant then so is  $\text{pre}(\tau, \varphi)$ , for every  $\tau \in \mathcal{T}$ .

**Justification** To establish the claim, we observe that

$$\{\varphi\} \tau \{\varphi\}: \underbrace{C_\tau \wedge \overline{V}' = \overline{E}}_{\rho_\tau} \wedge \varphi \rightarrow \varphi(\overline{V}')$$

is  $P$ -state valid if and only if

$$C_\tau \wedge \varphi \rightarrow \varphi[\overline{V} \mapsto \overline{E}],$$

which is equivalent to

$$\varphi \rightarrow \text{pre}(\tau, \varphi),$$

is  $P$ -state valid.

Assume that  $\varphi$  is  $P$ -invariant, i.e., it is  $P$ -state valid. By Problem 1.1, the verification condition  $\{\varphi\} \tau \{\varphi\}$ , which can also be written as  $\varphi \rightarrow \text{pre}(\tau, \varphi)$ , is  $P$ -state valid. It follows that  $\text{pre}(\tau, \varphi)$  is  $P$ -state valid, i.e., is  $P$ -invariant. ■

In **Problem 1.4**, we request the reader to compute the preconditions for example transitions and assertions.

The conjunction  $\varphi \wedge \text{pre}(\tau, \varphi)$  is obviously stronger than  $\varphi$  and is  $P$ -invariant whenever  $\varphi$  is. It has the added advantage stated by the following claim:

**Claim** For every assertion  $\varphi$  and program transition  $\tau$ , the verification condition

$$\{\varphi \wedge \text{pre}(\tau, \varphi)\} \tau \{\varphi \wedge \text{pre}(\tau, \varphi)\}$$

is state valid.

**Justification** Since  $\text{pre}(\tau, \varphi)$  has the form  $C_\tau \rightarrow \psi$  and all program transitions involve movement of control, it is easy to check that  $\rho_\tau$  implies  $\neg C'_\tau$ , and hence

implies  $\text{pre}'(\tau, \varphi) = (\text{pre}(\tau, \varphi))'$ . It is also easy to see that  $\rho_\tau \wedge \text{pre}(\tau, \varphi)$  implies  $\varphi'$ . We conclude that  $\varphi \wedge \text{pre}(\tau, \varphi)$  is preserved by  $\tau$  as stated in the claim. ■

Note that this claim is true also for the idling transition  $\tau_I$ , since  $\text{pre}(\tau_I, \varphi) = \varphi$  where the assertion  $\varphi$ , as well as any other assertion, is obviously preserved by  $\tau_I$ .

This leads to the following heuristic:

### Heuristic (supplement by preconditions)

If the verification condition  $\{\chi \wedge \varphi\} \tau \{\varphi\}$  is not state valid, strengthen  $\varphi$  by adding the conjunct  $\text{pre}(\tau, \varphi)$  or adding any assertion  $\psi$  that, under  $\chi$ , implies  $\text{pre}(\tau, \varphi)$ .

The beauty of this scheme is that it transforms weakness into strength. That is, we use an unprovable verification condition to obtain a supplementing assertion  $\text{pre}(\tau, \varphi)$  that may help us overcome the difficulty.

As previously discussed, strengthening an assertion by an additional conjunct, such as  $\text{pre}(\tau, \varphi)$ , can be utilized in two ways: incremental proof or direct strengthening. The incremental proof approach attempts to establish the invariance of  $\text{pre}(\tau, \varphi)$  in a separate step. Once  $\text{pre}(\tau, \varphi)$  is established as a  $P$ -invariant, we may repeat the proof of  $\varphi$ , using the stronger accumulated invariant  $\chi_{\text{new}}: \chi \wedge \text{pre}(\tau, \varphi)$  on the left-hand side of the premises. Note that, while re proving  $\varphi$ , it is not necessary to check the verification condition for  $\tau$ , the transition that determined  $\text{pre}(\tau, \varphi)$ . This is because  $\text{pre}(\tau, \varphi)$ , which implies the verification condition for  $\tau$  and  $\varphi$ , has just been shown to be an invariant.

In the direct strengthening approach, we apply rule INV with the stronger assertion  $\varphi_{\text{new}}: \varphi \wedge \text{pre}(\tau, \varphi)$ .

**Example** As an example for strengthening by incremental proofs, we resume our attempt to prove the invariance of

$$\varphi: \text{at\_}\ell_1 \rightarrow x = 2$$

for program ADD-TWO. As shown earlier,  $\varphi$  is not inductive. The offending transition is  $\ell_0$ . Consequently, we identified the supplementing assertion

$$\text{pre}(\ell_0, \varphi): \text{at\_}\ell_0 \rightarrow x = 0.$$

This assertion can be shown to be inductive and hence  $P$ -invariant. Repeating the attempt to verify  $\varphi$  by rule INV-B, where  $\chi$  is now

$$\chi: \text{at\_}\ell_0 \rightarrow x = 0,$$

we obtain the following verification condition for  $\varphi$

$$\underbrace{\text{move}(\ell_0, \ell_1) \wedge x' = x + 2}_{\rho_{\ell_0}} \wedge \underbrace{\text{at\_}\ell_0 \rightarrow x = 0}_{\chi} \wedge \underbrace{\dots}_{\varphi} \rightarrow \underbrace{\dots \rightarrow x' = 2}_{\varphi'},$$

which is state valid since  $\text{move}(\ell_0, \ell_1)$  implies  $\text{at\_}\ell_0$  which, by  $\chi$ , yields  $x = 0$ . As commented earlier, there is actually no need to reprove this verification condition, since the supplementing assertion  $\chi: \text{at\_}\ell_0 \rightarrow x = 0$  is derived from  $\text{pre}(\ell_0, \varphi)$ .

Direct strengthening applies rule INV, using the strengthened (inductive) assertion

$$\underbrace{\text{at\_}\ell_1 \rightarrow x = 2}_{\varphi} \wedge \underbrace{\text{at\_}\ell_0 \rightarrow x = 0}_{\chi}. \blacksquare$$

### Example (summation)

As a more advanced example, reconsider program SUM of Fig. 1.12, for which we wish to establish the invariance of the assertion

$$\varphi_2: \text{at\_}\ell_2 \rightarrow \text{sum} = \sum_{r=1}^N A[r].$$

This assertion is not inductive. Its verification condition for  $\ell_0^F$  is not state valid. Consequently, we form the precondition  $\text{pre}(\ell_0^F, \varphi_2)$  which, after simplification, yields

$$\text{at\_}\ell_0 \wedge i = N \rightarrow \text{sum} = \sum_{r=1}^N A[r].$$

As a strengthening conjunct we can add either this precondition or an assertion that implies it. We choose to add the conjunct

$$\varphi_0: \text{at\_}\ell_0 \rightarrow \text{sum} = \sum_{r=1}^i A[r].$$

In checking the conjunction  $\varphi_0 \wedge \varphi_2$  for inductiveness, we find that its verification condition for transition  $\ell_1$  is not state valid. Consequently, we form the precondition  $\text{pre}(\ell_1, \varphi_0 \wedge \varphi_2)$ , which yields

$$\begin{aligned} \forall V': \underbrace{\text{move}(\ell_1, \ell_0) \wedge i' = i + 1 \wedge \text{sum}' = \text{sum} + A[i + 1]}_{\rho_{\ell_1}} \rightarrow \\ \underbrace{\text{at'}_{\ell_0} \rightarrow \text{sum}' = \sum_{r=1}^{i'} A[r]}_{\varphi'_0} \wedge \underbrace{\text{at'}_{\ell_2} \rightarrow \dots}_{\varphi'_2}. \end{aligned}$$

Applying the standard simplification by substitution, and using  $\text{move}(\ell_1, \ell_0)$  to infer  $\text{at'}_{\ell_0} = \top$  and  $\text{at'}_{\ell_2} = \perp$ , this reduces to

$$\text{at\_}\ell_1 \rightarrow \text{sum} + A[i + 1] = \sum_{r=1}^{i+1} A[r],$$

which can be simplified to

$$\varphi_1: \text{at-}\ell_1 \rightarrow \text{sum} = \sum_{r=1}^i A[r].$$

Taking the conjunction of the three assertions, we obtain the assertion

$$\varphi: \left( \text{at-}\ell_{0,1} \rightarrow \text{sum} = \sum_{r=1}^i A[r] \right) \wedge \left( \text{at-}\ell_2 \rightarrow \text{sum} = \sum_{r=1}^N A[r] \right),$$

which is inductive and implies  $\varphi_2$ . ■

The technique of strengthening assertion  $\varphi$  by supplementing it with preconditions that are not implied by  $\varphi$  (or by  $\chi \wedge \varphi$ ) is often referred to as *assertion propagation*.

Consider, for example, program SUM. The assertion with which we started,

$$\text{at-}\ell_2 \rightarrow \text{sum} = \sum_{r=1}^N A[r],$$

gives us meaningful information only when control is at  $\ell_2$ . The first strengthening step supplements this assertion by the implication

$$\text{at-}\ell_0 \rightarrow \text{sum} = \sum_{r=1}^i A[r],$$

which provides information about the value of *sum* when control is at  $\ell_0$ . We can view this step as the propagation of information from  $\ell_2$  back to  $\ell_0$ , along mode  $\ell_0^F$ . The next supplementation propagates information from  $\ell_0$  back to  $\ell_1$  via transition  $\ell_1$  and yields the missing link

$$\text{at-}\ell_1 \rightarrow \text{sum} = \sum_{r=1}^i A[r].$$

We will further illustrate the use of this important heuristic on additional examples.

### Example (integer square root)

For program SQUARE-ROOT (Fig. 1.11), the main safety property that we wish to establish is that of partial correctness. Partial correctness for this program is given by the invariance of

$$\psi: \text{at-}\ell_2 \rightarrow (z^2 \leq x \wedge x < (z+1)^2)$$

This specification consists of two parts which assure that, on termination, both  $z^2 \leq x$  and  $x < (z+1)^2$  hold. The invariance of the second part,

$$\psi_2: \text{at-}\ell_2 \rightarrow x < (z+1)^2,$$

was previously established, using the two invariants

$$\varphi_1: u = 2 \cdot z + 1$$

$$\varphi_2: w = (z + 1)^2.$$

The invariance of the first part,

$$\psi_1: at\_\ell_2 \rightarrow z^2 \leq x,$$

cannot be verified by rule INV-B using the invariants  $\varphi_1$  and  $\varphi_2$ . We therefore need a stronger version of  $\psi_1$  that can be proven to be an invariant.

- *Strengthening  $\psi_1$ :*

As a first step, we remove the range restriction  $at\_\ell_2$  from  $\psi_1$  and obtain the stronger assertion

$$\hat{\psi}_1: z^2 \leq x.$$

Checking whether this assertion is inductive, we form the precondition for  $\hat{\psi}_1$  with respect to  $\ell_1$

$$\Phi_1 = pre(\ell_1, \hat{\psi}_1): at\_\ell_1 \rightarrow (z + 1)^2 \leq x.$$

This assertion is not implied by  $\varphi_1 \wedge \varphi_2$ . We therefore adopt  $\Phi_1$  as a supplementing assertion and consider

$$z^2 \leq x \wedge (at\_\ell_1 \rightarrow (z + 1)^2 \leq x),$$

which can also be represented, using arithmetization, as

$$\varphi_3: (z + at\_\ell_1)^2 \leq x.$$

- *Establishing  $\varphi_3$ :*

Initially  $at\_\ell_1 = F$  and  $z = 0$ , and since the initial condition of the program ensures  $x \geq 0$ , it follows that initially  $(z + at\_\ell_1)^2 \leq x$ .

Consider the verification condition of  $\varphi_3$  over modes  $\ell_0^F$ ,  $\ell_0^T$ , and transition  $\ell_1$ .

Mode  $\ell_0^F$  does not change the variables  $z$  or  $x$ , or the control expression  $at\_\ell_1$  on which  $\varphi_3$  depends. Consequently, it certainly preserves  $\varphi_3$ .

The verification condition for  $\ell_0^T$  is given by

$$\rho_0^T \wedge (z + at\_\ell_1)^2 \leq x \rightarrow (z + at'\_\ell_1)^2 \leq x.$$

The transition relation  $\rho_0^T$  implies  $at'\_\ell_1 = 1$  and  $w \leq x$  which, using the invariance of  $\varphi_2$ :  $w = (z + 1)^2$ , establishes the validity of the verification condition.

Next, consider the transition  $\ell_1$ . Here we have to show

$$\rho_1 \wedge (z + at\_\ell_1)^2 \leq x \rightarrow (z' + at'\_\ell_1)^2 \leq x.$$

Since  $\rho_1$  implies  $at\_\ell_1 = 1$ ,  $at'\_\ell_1 = 0$ , and  $z' = z + 1$ , it is sufficient to show

$$(z + 1)^2 \leq x \rightarrow ((z + 1) + 0)^2 \leq x,$$

which is trivially valid.

Note that, in fact, there was no need to reconsider the transition  $\ell_1$ . This is because the supplementing assertion  $\Phi_1$  was designed to guarantee the verification condition for  $\ell_1$ .

Thus, the inductiveness of  $\varphi_3$  (relative to  $\varphi_2$ ) has been established.

- *Establishing  $\psi$ :*

Since  $\varphi_3$  is inductive, it is invariant. As  $\varphi_3$  implies  $\psi_1$ , it follows that  $\psi_1$  is also invariant. ■

The next example combines bottom-up and top-down methods to prove invariance properties of an algorithm that implements mutual exclusion without semaphores.

### **Example** (Peterson's algorithm for mutual exclusion)

Consider program MUX-PET1 presented in Fig. 1.13, which implements Peterson's algorithm for mutual exclusion.

The basic mechanism protecting the accesses to the critical sections is provided by the boolean variables  $y_1$  and  $y_2$ . Each process  $P_i$ ,  $i = 1, 2$ , that is interested in entering its critical section sets its  $y_i$  variable to T. On exiting the critical section, the corresponding  $y_i$  is reset to F.

The problem with this simple-minded approach is that the two processes may arrive at their waiting positions,  $\ell_3$  and  $m_3$ , at about the same time, with both  $y_1 = y_2 = \text{T}$ . If the only criterion for entry to the critical section were that the  $y_i$  of the competitor is F, this situation would result in a deadlock (tie).

The variable  $s$  is intended to break such ties. It may be viewed as a *signature*, in the sense that each process that sets its  $y_i$  variable to T also writes its identity number in  $s$  at the same step. Thus, if both processes are at the waiting position, the first to enter will be  $P_i$  such that  $s \neq i$ . For  $i = 1, 2$ , let  $j$  denote the index of the other process. The fact that  $s \neq i$  implies that  $s = j$ , which means that the competitor,  $P_j$ , was the *last* to set its variable  $y_j$  to T, and therefore  $P_i$  should have priority.

The main invariance property of this program is mutual exclusion, which can be expressed by the invariance of

$$\psi: \neg(at\_l_4 \wedge at\_m_4).$$

We start by establishing some bottom-up invariances. The following assertions are inductive and easy to check.

$$\varphi_0: s = 1 \vee s = 2$$

$$\varphi_1: y_1 \leftrightarrow at\_l_{3.5}$$

**local**  $y_1, y_2$ : boolean where  $y_1 = \text{F}$ ,  $y_2 = \text{F}$   
 $s$  : integer where  $s = 1$

$$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: (y_1, s) := (\text{T}, 1) \\ \ell_3: \text{await } \neg y_2 \vee s \neq 1 \\ \ell_4: \text{critical} \\ \ell_5: y_1 := \text{F} \end{array} \right] \end{array} \right]$$

$$||$$

$$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: (y_2, s) := (\text{T}, 2) \\ m_3: \text{await } \neg y_1 \vee s \neq 2 \\ m_4: \text{critical} \\ m_5: y_2 := \text{F} \end{array} \right] \end{array} \right]$$

Fig. 1.13. Program MUX-PETI1 (Peterson's algorithm for mutual exclusion) — version 1.

$$\varphi_2: y_2 \leftrightarrow at\_m_{3..5}.$$

Invariant  $\varphi_0$  is inspired by the observation that  $s$  is initially 1 and is only assigned the constants 1 or 2 in the program. Invariant  $\varphi_1$  is inspired by the observation that variable  $y_1$  is set to T by the assignment  $\ell_2$  and reset to F by assignment  $\ell_5$ . A similar observation leads to  $\varphi_2$ .

Next, we concentrate on proving  $\psi: \neg(at\_\ell_4 \wedge at\_m_4)$ . Assertion  $\psi$  by itself is not inductive, even when we take  $\varphi_0$ ,  $\varphi_1$ , and  $\varphi_2$  into account. The two offending transitions are  $\ell_3$  and  $m_3$ .

Consider the precondition of  $\psi$  with respect to  $\ell_3$ . It is given by

$$\forall \pi': \underbrace{\text{move}(\ell_3, \ell_4) \wedge (\neg y_2 \vee s \neq 1)}_{\rho_{\ell_3}} \rightarrow \underbrace{\neg(at'\_\ell_4 \wedge at'\_m_4)}_{\psi'}$$

which, observing that  $\text{move}(\ell_3, \ell_4)$  implies  $at'\_\ell_4 = \text{T}$  and  $at'\_m_4 = at\_m_4$ , reduces to

$$at\_l_3 \wedge (\neg y_2 \vee s \neq 1) \rightarrow \neg at\_m_4$$

which is equivalent to

$$\varphi_3: at\_l_3 \wedge at\_m_4 \rightarrow y_2 \wedge s = 1.$$

In a symmetric way, we form  $pre(m_3, \psi)$  and show that it is equivalent to

$$\varphi_4: at\_l_4 \wedge at\_m_3 \rightarrow y_1 \wedge s = 2.$$

It is not difficult to check that all other preconditions  $pre(\tau, \psi)$ , for  $\tau \notin \{\tau_1, l_3, m_3\}$  are trivially true since these transitions lead to either a false  $at'_l l_4$  or a false  $at'_l m_4$ . Consequently,  $\psi$  is inductive relative to  $\varphi_3 \wedge \varphi_4$ , and proving the invariance of  $\varphi_3$  and  $\varphi_4$  establishes the invariance of  $\psi$ .

We proceed to show that  $\varphi_3$  and  $\varphi_4$  are inductive relative to  $\varphi_0$ ,  $\varphi_1$ , and  $\varphi_2$ . Since  $\varphi_3$  and  $\varphi_4$  are symmetric, we will only verify  $\varphi_3$ .

- *Establishing  $\varphi_3$ :*

Let us show that  $\varphi_3$  is inductive. Trivially,  $\varphi_3$  holds initially, since then  $at\_l_3 = at\_m_4 = \text{f}$ . The transitions that we should check are those that validate  $at\_l_3 \wedge at\_m_4$  and those that may falsify  $y_2 \wedge s = 1$ .

The transitions that validate  $at\_l_3 \wedge at\_m_4$  are:

- $l_2$  while  $at\_m_4$  holds.

It is sufficient to show

$$\underbrace{\dots \wedge s' = 1}_{\rho_{l_2}} \wedge at\_m_4 \rightarrow y_2 \wedge s' = 1.$$

Due to  $\varphi_2$ ,  $at\_m_4$  implies  $y_2 = \text{t}$ . The transition  $l_2$  establishes  $s' = 1$ .

- $m_3$  while  $at\_l_3$  holds.

It is sufficient to show

$$\underbrace{\dots \wedge (\neg y_1 \vee s \neq 2) \wedge \dots}_{\rho_{m_3}} \wedge at\_l_3 \rightarrow y_2 \wedge s = 1.$$

Again, due to  $\varphi_2$  and  $at\_m_3$ ,  $y_2 = \text{t}$ . Using  $\varphi_1$ , when  $at\_l_3$  holds,  $y_1 = \text{t}$ . Therefore,  $m_3$  is enabled only when  $s \neq 2$ . Due to  $\varphi_0$ , this implies  $s = 1$ .

The only transitions that may falsify  $y_2 \wedge s = 1$  are  $m_2$  and  $m_5$ . The transition relations of both transitions imply that  $at\_m_4$  is false after the transition. Consequently, these transitions also preserve  $\varphi_3$ .

This concludes the proof of mutual exclusion for program MUX-PET1. ■

In Problems 1.5–1.8, we ask the reader to establish partial correctness for some sequential terminating programs.

**Example** (refined version of Peterson's algorithm)

Peterson's algorithm, as presented in Fig. 1.13, is one of the simplest algorithms for mutual exclusion. Unfortunately, the version of Fig. 1.13 is not directly implementable on conventional hardware because statement  $\ell_2$  (similarly  $m_2$ ) requires assignment of values to the two variables  $y_1$  and  $s$  at the same time (in one atomic step). In our terminology, we complain that program MUX-PET<sub>1</sub> is not an LCR-program since statement  $\ell_2$  (and  $m_2$ ) contains two critical references.

In Fig. 1.14 we present program MUX-PET<sub>2</sub>, which is an LCR-program. Program MUX-PET<sub>2</sub> can be viewed as a refined version of program MUX-PET<sub>1</sub> obtained by decomposing statement  $\ell_2$  of MUX-PET<sub>1</sub> into the consecutive assignments  $\ell_2$  and  $\ell_3$  in MUX-PET<sub>2</sub> and applying a similar decomposition to statement  $m_2$  of MUX-PET<sub>1</sub>. We also decomposed  $\ell_3$  ( $m_3$ ) into a selection between statements  $\ell_4^a$  and  $\ell_4^b$  ( $m_4^a$  and  $m_4^b$ ) in MUX-PET<sub>2</sub>. Obviously, MUX-PET<sub>2</sub> is an LCR-program and can therefore be implemented on conventional hardware.

In general, decomposing a multiple assignment into a concatenation of single assignments may transform a correct program into an incorrect one. Therefore, we cannot simply use our previous proof that MUX-PET<sub>1</sub> maintains mutual exclusion to conclude that MUX-PET<sub>2</sub> does as well. However, we may be guided by the list of invariants generated in the verification of MUX-PET<sub>1</sub> in our search for similar invariants for MUX-PET<sub>2</sub>, which may be viewed as refinements of corresponding MUX-PET<sub>1</sub> invariants.

The invariants we derived for program MUX-PET<sub>1</sub> are the following:

$$\begin{aligned}\chi_0: \quad & s = 1 \vee s = 2 \\ \chi_1: \quad & y_1 \leftrightarrow \text{at\_}\ell_{3..5} \\ \chi_2: \quad & y_2 \leftrightarrow \text{at\_}\boldsymbol{m}_{3..5} \\ \chi_3: \quad & \text{at\_}\ell_3 \wedge \text{at\_}\boldsymbol{m}_4 \rightarrow y_2 \wedge s = 1 \\ \chi_4: \quad & \text{at\_}\ell_4 \wedge \text{at\_}\boldsymbol{m}_3 \rightarrow y_1 \wedge s = 2 \\ \chi_5: \quad & \neg(\text{at\_}\ell_4 \wedge \text{at\_}\boldsymbol{m}_4).\end{aligned}$$

We can inspect each of these assertions as a candidate for an invariant of program MUX-PET<sub>2</sub>. In some of them, we have to allow for a shift in label numbers since what MUX-PET<sub>1</sub> accomplished in one statement requires two statements in program MUX-PET<sub>2</sub>.

Assertion  $\chi_0$  can easily be shown to be inductive also for MUX-PET<sub>2</sub>. We therefore list it as the first established invariant

$$\varphi_0: \quad s = 1 \vee s = 2.$$

When considering  $\chi_1$  and  $\chi_2$ , we have to extend the location ranges to  $\ell_6$  and  $m_6$ . It is straightforward to establish that the following two extensions of  $\chi_1$  and  $\chi_2$  are inductive over MUX-PET<sub>2</sub>:

```
local  $y_1, y_2$ : boolean where  $y_1 = F, y_2 = F$ 
 $s$  : integer where  $s = 1$ 
```

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: y_1 := T \\ \ell_3: s := 1 \\ \ell_4: \left[ \begin{array}{l} \ell_4^a: \text{await } \neg y_2 \\ \text{or} \\ \ell_4^b: \text{await } s \neq 1 \end{array} \right] \\ \ell_5: \text{critical} \\ \ell_6: y_1 := F \end{array} \right] \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: y_2 := T \\ m_3: s := 2 \\ m_4: \left[ \begin{array}{l} m_4^a: \text{await } \neg y_1 \\ \text{or} \\ m_4^b: \text{await } s \neq 2 \end{array} \right] \\ m_5: \text{critical} \\ m_6: y_2 := F \end{array} \right] \end{array} \right]$

Fig. 1.14. Program MUX-PET2 (Peterson's algorithm)  
— refined version.

$$\varphi_1: y_1 \leftrightarrow \text{at-}\ell_{3..6}$$

$$\varphi_2: y_2 \leftrightarrow \text{at-}m_{3..6}.$$

Next, let us consider assertions  $\chi_3$  and  $\chi_4$ . Obviously, the references in  $\chi_3$  to locations  $\ell_3$  and  $m_4$  are intended to point to the location just before the critical section in  $P_1$  and to the first critical location in  $P_2$ . For program MUX-PET2 these are given by  $\ell_4$  and  $m_5$ , respectively. Consequently, we formulate the following assertion as a candidate for an invariant:

$$\varphi_3: \text{at\_}\ell_4 \wedge \text{at\_}m_5 \rightarrow y_2 \wedge s = 1.$$

Using invariants  $\varphi_0$ ,  $\varphi_1$ , and  $\varphi_2$ , we can show that  $\varphi_3$  is inductive (relative to  $\varphi_0$ ,  $\varphi_1$ ,  $\varphi_2$ ).

In a symmetric way, we can formulate and prove the inductiveness of the following assertion

$$\varphi_4: \text{at\_}\ell_5 \wedge \text{at\_}m_4 \rightarrow y_1 \wedge s = 2.$$

We can now formulate and prove the final assertion stating mutual exclusion for program MUX-PET2

$$\varphi_5: \neg(\text{at\_}\ell_5 \wedge \text{at\_}m_4).$$

It is straightforward to verify that  $\varphi_5$  is inductive relative to  $\varphi_3$  and  $\varphi_4$ . In particular, the verification conditions  $\{\varphi_5\} \ell_4^a \{\varphi_5\}$  and  $\{\varphi_5\} \ell_4^b \{\varphi_5\}$  follow from  $\varphi_3$  which ensures that when  $P_2$  is at  $m_5$ , both  $\ell_4^a$  and  $\ell_4^b$  are disabled. In a similar way, the preservation of  $\varphi_5$  over  $m_4^a$  and  $m_4^b$  follows from  $\varphi_4$ .

This concludes the proof of mutual exclusion for program MUX-PET2. ■

### Example (Dekker's algorithm)

One of the first correct algorithms to solve the mutual-exclusion problem was the one suggested by Dekker. The basic idea in this algorithm is simple and similar to that of Peterson's, which was developed several years later. Each of the processes  $P_i$  has a boolean variable  $y_i$  that expresses the interest of the process in entering its critical section. If one of the processes, after setting its variable  $y_i$  to T, finds that variable  $y_j$  of its competitor is false, it enters the critical section immediately. In case of a tie, i.e., both processes have a true  $y_i$ , we use a tie-breaker, the variable  $t$  (short for *turn*). This variable ranges over  $\{1, 2\}$ , and in case of a tie, the process whose index equals  $t$  has the higher priority. To ensure fair accessibility, each process sets variable  $t$  to the index of its rival on exit from the critical section.

The algorithm based on these ideas is presented in program MUX-DEK of Fig. 1.15.

Let us follow  $P_1$  on its departure from the noncritical section. This is where the protocol of coordination between the two processes starts. Process  $P_1$  sets variable  $y_1$  to T at  $\ell_2$ . It then enters a *while* loop that continues as long as  $P_1$  detects a true  $y_2$ . When  $P_1$  detects a true  $y_2$  it identifies the situation as a tie. Tie breaking is accomplished by one of the processes recognizing that it has a lower priority, resetting its variable  $y_i$  to F, and then waiting for its priority to rise. This happens for  $P_1$  in  $\ell_5-\ell_7$ . On the other hand, if  $P_1$  recognizes it has a higher priority, i.e.  $t = 1$ , it leaves its  $y_1$  variable true and waits for  $y_2$  to become false. This happens for  $P_1$  in the loop consisting of  $\ell_3, \ell_4$ . The process  $P_1$  enters its critical section at  $\ell_8$  only when it detects a false  $y_2$ . After termination of the

```

local y1, y2: boolean where y1 = F, y2 = F
t      : integer where t = 1

P1 :: [l0: loop forever do
          [l1: noncritical
           l2: y1 := T
           l3: while y2 do
           l4: if t = 2
                  then [l5: y1 := F
                         l6: await t = 1
                         l7: y1 := T]
           l8: critical
           l9: t := 2
           l10: y1 := F]
        ]
||

P2 :: [m0: loop forever do
          [m1: noncritical
           m2: y2 := T
           m3: while y1 do
           m4: if t = 1
                  then [m5: y2 := F
                         m6: await t = 2
                         m7: y2 := T]
           m8: critical
           m9: t := 1
           m10: y2 := F]
        ]

```

Fig. 1.15. Program MUX-DEK (Dekker's algorithm for mutual exclusion).

critical section, P<sub>1</sub> first sets t to 2 and then resets y<sub>1</sub> to F.

In comparison to Peterson's algorithm (program MUX-PET1), Dekker's algorithm (program MUX-DEK) has a relatively simple safety proof.

For program MUX-DEK, we can derive the following invariants:

$$\begin{aligned}\varphi_0: \quad & t = 1 \vee t = 2 \\ \varphi_1: \quad & y_1 \leftrightarrow (\text{at-}l_{3..5,8..10}) \\ \varphi_2: \quad & y_2 \leftrightarrow (\text{at-}m_{3..5,8..10}).\end{aligned}$$

In **Problem 1.9** the reader is requested to establish these invariants and prove the mutual-exclusion property for program MUX-DEK, i.e., that

$$\psi: \neg(\text{at-}l_{8..10} \wedge \text{at-}m_{8..10}),$$

is also invariant. ■

In **Problem 1.10**, the reader is requested to prove mutual exclusion for another program.

## 1.5 Refining Invariants

As the reader may have observed, some of the example programs presented in the preceding sections have a coarse granularity, using large grouped statements. Considering the complexity of the verification process, this is always a good idea, since it leads to a smaller number of transitions and, hence, to fewer verification conditions and control configurations.

When we deal with sequential programs, coarser granularity is associated neither with high implementation costs nor with restriction of the set of behaviors. Therefore, there is no reason for refraining from taking the coarsest granularity possible. For concurrent programs, however, coarser granularity is associated with higher costs of implementation. This is because the implementation of grouped statements necessitates the use of semaphores or similar synchronization constructs to ensure no interference in the execution of statements of the group. These additional synchronization and locking constructs introduce unnecessary restraints on the independent operation of parallel components in the system, and may significantly reduce efficient use of parallelism. Therefore, while verification considerations recommend coarser granularity, efficiency considerations favor finer granularity.

An acceptable compromise is to first perform a preliminary analysis of a coarser version of a program and verify its correctness. Then, derive a finer version of the program by refining large grouped statements into smaller statements. Since refinement of concurrent programs introduces, in general, new computations, the finer version still has to be verified. However, as we show in this section, it is possible to use the invariant assertions used in the verification of the coarse version to derive invariant assertions for the finer version. We refer to this derivation as the *refinement of invariants*.

A similar strategy may be suggested for the development of programs. We

first develop a coarse-granularity program that satisfies the specification. Then, to obtain a more efficient program, we refine the coarse program into a finer one. This refinement step has to be justified by techniques similar to the ones we study in this section.

## Filling the Gaps

One of the effects of program refinement is that the refined program contains more control locations than the original program. In many cases, we may assume that the invariant assertions established for the coarse program are still valid in the refined program whenever the control configuration coincides with one that exists in the coarse program. Then, we employ a propagation technique to find the assertions that hold in the configurations that exist only in the newly refined program.

### Example (integer square root)

Consider, for example, program **SQUARE-ROOT\*** of Fig. 1.16 for computing the integer square root of a nonnegative integer. This program is a relabeled version of program of **SQUARE-ROOT** of Fig. 1.11, in which label  $\ell_2$  is relabeled  $\ell_4$ .

```
in  x  : integer where  $x \geq 0$ 
local u, w: integer where  $u = 1, w = 1$ 
out z  : integer where  $z = 0$ 
```

$\ell_0$ : **while**  $w \leq x$  **do**

$\ell_1$ :  $(z, u, w) := (z + 1, u + 2, w + u + 2)$

$\ell_4$ :

Fig. 1.16. Program **SQUARE-ROOT\*** (integer square root)  
— coarse version.

Program **SQUARE-ROOT-R** in Fig. 1.17 can be viewed as a refinement of program **SQUARE-ROOT\*** of Fig. 1.16, where the single multiple assignment statement  $\ell_1$  of Fig. 1.16 has been refined into the statements  $\ell_1, \ell_2$ , and  $\ell_3$  in Fig. 1.17. Note that the expression assigned to  $w$  has been modified to refer to the new value of  $u$ .

The refined program has the locations  $\ell_0, \ell_1, \ell_2, \ell_3$ , and  $\ell_4$ . Of these,  $\ell_0, \ell_1$ , and  $\ell_4$  already exist in the coarse program, while  $\ell_2$  and  $\ell_3$  are newly introduced.

The invariant assertion established in Sections 1.3 and 1.4 for the coarse program of Fig. 1.16 is

$$\varphi_1 \wedge \varphi_2 \wedge \varphi_3: \quad u = 2 \cdot z + 1 \quad \wedge \quad w = (z + 1)^2 \quad \wedge \quad (z + at_{-\ell_1})^2 \leq x.$$

We hope that this assertion holds whenever we visit one of the old locations, which leads to the following candidate invariant:

$$\varphi: \quad at_{-\ell_{0,1,4}} \rightarrow \varphi_1 \wedge \varphi_2 \wedge \varphi_3.$$

```

in   x   : integer where x ≥ 0
local u, w: integer where u = 1, w = 1
out  z   : integer where z = 0

ℓ₀: while w ≤ x do
    [ℓ₁: z := z + 1
     ℓ₂: u := u + 2
     ℓ₃: w := w + u]

ℓ₄:

```

Fig. 1.17. Program SQUARE-ROOT-R — refined version.

Since several new transitions and locations have been introduced, this assertion is certainly not inductive. We use propagation techniques for finding the assertions that provide information at the new locations.

- *Propagation to  $\ell_2$  and  $\ell_3$ :*

As a first step, we construct the precondition with respect to  $\ell_3$ . This is given by

$$\forall \pi', w': \underbrace{move(\ell_3, \ell_0) \wedge w' = w + u \wedge z' = z \wedge u' = u}_{\rho_{\ell_3}} \rightarrow$$

$$at'_{-\ell_{0,1,4}} \rightarrow \underbrace{u' = 2 \cdot z' + 1 \wedge w' = (z' + 1)^2 \wedge (z' + at'_{-\ell_1})^2 \leq x}_{\varphi'}$$

which, observing that  $move(\ell_3, \ell_0)$  implies  $at'_{-\ell_{0,1,4}} = \top$  and  $at'_{-\ell_1} = \perp$ , simplifies to

$$at_{-\ell_3} \rightarrow u = 2 \cdot z + 1 \wedge w + u = (z + 1)^2 \wedge z^2 \leq x.$$

The second conjunct of the right-hand side conjunction can be expanded to

$$w + u = z^2 + 2 \cdot z + 1,$$

which can be simplified using the first conjunct to  $w = z^2$ . We therefore have

$$\Phi_1: \text{at-}\ell_3 \rightarrow u = 2 \cdot z + 1 \wedge w = z^2 \wedge z^2 \leq x.$$

We proceed to form the precondition of  $\varphi \wedge \Phi_1$  with respect to  $\ell_2$ . Observing that the data transformation for  $\ell_2$  is given by  $u' = u + 2$ , and that  $\text{move}(\ell_2, \ell_3)$  implies  $\text{at}'\ell_3 = \top$  and  $\text{at}'\ell_{0,1,4} = \text{F}$ , we obtain the assertion

$$\Phi_2: \text{at-}\ell_2 \rightarrow u + 2 = 2 \cdot z + 1 \wedge w = z^2 \wedge z^2 \leq x.$$

This completes the construction of the full assertion  $\widehat{\varphi}: \varphi \wedge \Phi_1 \wedge \Phi_2$ , which can be written as

$$\widehat{\varphi}: u + 2 \cdot \text{at-}\ell_2 = 2 \cdot z + 1 \wedge w = (z + \text{at-}\ell_{0,1,4})^2 \wedge (z + \text{at-}\ell_1)^2 \leq x.$$

- *Establishing the invariance of  $\widehat{\varphi}$ :*

To check the inductiveness of  $\widehat{\varphi}$ , we have only to check that

$$\Theta \rightarrow \widehat{\varphi},$$

and the verification condition for  $\ell_1$ . This is because transitions  $\ell_2$  and  $\ell_3$  have been used in the propagation process that derived  $\widehat{\varphi}$  from  $\varphi$ . The verification conditions of  $\varphi$  with respect to  $\ell_0^\top$  and  $\ell_0^\text{F}$  have already been checked in the context of the coarse version of the program. Thus, the only remaining transition is  $\ell_1$ .

The initial condition  $\Theta$  is given by

$$\Theta: \pi = \{\ell_0\} \wedge x \geq 0 \wedge z = 0 \wedge u = 1 \wedge w = 1,$$

which obviously implies  $\widehat{\varphi}$ .

The verification condition for  $\ell_1$  is

$$\rho_{\ell_1} \wedge \widehat{\varphi} \rightarrow \widehat{\varphi}'.$$

Since  $\rho_{\ell_1}$  implies  $\text{at-}\ell_1$ ,  $\text{at}'\ell_2$ , and  $z' = z + 1$ , it is sufficient to check

$$z' = z + 1 \wedge u = 2 \cdot z + 1 \wedge w = (z + 1)^2 \wedge (z + 1)^2 \leq x \rightarrow \\ u + 2 = 2 \cdot z' + 1 \wedge w = (z')^2 \wedge (z')^2 \leq x.$$

Clearly,  $z' = z + 1 \wedge u = 2 \cdot z + 1$  implies  $u + 2 = 2 \cdot z' + 1$ . The rest is simple.

Thus, by rule INV-B,  $\widehat{\varphi}$  is indeed an invariant of the refined version of the program. ■

In Problem 1.11, we ask the reader to prove partial correctness of a program that computes the cubic power of an integer using additions only. In Problems 1.12–1.16, the reader is requested to verify mutual exclusion for several algorithms, some of which use special coordination statements.

## Compensation Expressions

A drawback of the propagation technique is that it forces us to enumerate many distinct control configurations that can arise in the computations of a program. For a concurrent program this number can be very large. A somewhat different approach to the refinement of invariant assertions is provided by *compensation expressions* and may lead to assertions of smaller size.

### Example (producer-consumer)

Let us consider PROD-CONS-C, a coarse version of the asynchronously communicating producer-consumer program, presented in Fig. 1.18. The program consists of a producer process *Prod* and a consumer process *Cons*. The producer alternates between a production activity, represented by the schematic statement **produce**  $x$ , and a transmission activity performed in the grouped statement  $\ell_2$ . The consumer alternates between a receiving activity at statement  $m_1$  and a consuming activity, represented by the schematic statement **consume**  $y$ .

**local** *send, ack*: channel [1..] of integer  
**where** *send* =  $\Lambda$ , *ack* =  $\underbrace{[1, \dots, 1]}_N$

$Prod :: \left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \langle ack \Rightarrow t \\ \qquad \qquad \qquad send \Leftarrow x \rangle \end{array} \right] \end{array} \right] \parallel Cons :: \left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \langle send \Rightarrow y \\ \qquad \qquad \qquad ack \Leftarrow 1 \rangle \\ m_3: \text{consume } y \end{array} \right] \end{array} \right]$	
---	--

Fig. 1.18. Program PROD-CONS-C (producer-consumer) — coarse version.

This program represents the situation that channel *send* is unbounded, yet we wish to guarantee that it never holds more than  $N$  messages at a time. This can correspond to the situation that an attempt to send an additional message to channel *send* while it already holds  $N$  messages may cause the system to fail.

To ensure that the transmission channel *send* never holds more than  $N$  pending messages, the program uses an acknowledgement channel *ack* that initially contains  $N$  messages having the value 1. Messages in channel *ack* are to be viewed as permissions to send a message via channel *send*. Thus, before sending  $x$  via *send*, the producer must remove one permission from *ack*. Similarly, after

reading (and removing) a message from *send*, the consumer places a permission on channel *ack*.

In the initial state, channel *ack* holds  $N$  “permissions.” The number  $N$  is a parameter of the program, which is assumed to be positive.

- *Coarse version*

The coarse program of Fig. 1.18 reads from *ack* and writes to *send* in one transition at  $\ell_2$ , and similarly reads from *send* and writes to *ack* in one transition at  $m_1$ .

An important safety property of program PROD-CONS-C is that the *send* channel never contains more than  $N$  messages (values). This is expressed by the invariance of the assertion

$$\psi: \quad |\text{send}| \leq N.$$

A simple bottom-up analysis of this program observes that channels *ack* and *send* are always jointly modified, and that their modification is such that it preserves the value of the expression  $|\text{ack}| + |\text{send}|$ , which represents the sum of the lengths of the two channel buffers. Since the initial value of this expression is  $N$ , we propose as invariant the assertion

$$\varphi: \quad |\text{ack}| + |\text{send}| = N.$$

It is not difficult to check that this assertion is indeed inductive and, because  $|\text{ack}| \geq 0$  by definition,  $\varphi$  implies the invariance of  $\psi$ .

This establishes the validity of the desired safety property for program PROD-CONS-C, the coarse version of program PROD-CONS.

- *Refined version*

Next, we consider a refined version of the program in which we have split the two grouped statements  $\ell_2$  and  $m_1$  into their substatements. The refined version, Program PROD-CONS, is presented in Fig. 1.19. Note that there are two new control locations,  $\ell_3$  and  $m_2$ , which are present in the refined version but not in the coarse one. It is not difficult to see that assertion  $\varphi$  does not hold when control is in either of the new locations.

The assertion  $\varphi$  holds for the coarse version of the program since, in that version, the channels *ack* and *send* are always jointly modified by the same transitions, and this modification preserves the sum  $|\text{ack}| + |\text{send}|$ . In the fine version, these two channels are modified by separate (though consecutive) transitions. Thus, when control is at  $\ell_3$ , one permission has been removed from channel *ack*, but the message to channel *send* has not yet been sent. Consequently, when control is at  $\ell_3$ , the value of the expression  $|\text{send}| + |\text{ack}|$  has a deficit of 1, relative to the value it would have had if  $\ell_2$  and  $\ell_3$  were executed as one atomic transition. This deficit is paid back when we actually perform the statement at  $\ell_3$ .

**local** *send, ack*: channel [1..] of integer

**where** *send* =  $\Lambda$ , *ack* =  $\underbrace{[1, \dots, 1]}_N$

*Prod* ::  $\left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \text{ack} \Rightarrow t \\ \ell_3: \text{send} \Leftarrow x \end{array} \right] \end{array} \right] \parallel \text{Cons} :: \left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{send} \Rightarrow y \\ m_2: \text{ack} \Leftarrow 1 \\ m_3: \text{consume } y \end{array} \right] \end{array} \right]$

Fig. 1.19. Program PROD-CONS — refined version.

A similar situation holds at  $m_2$ . Here, a message has been removed from *send*, and a corresponding permission not yet placed on *ack*. Consequently, there is another deficit of 1 associated with being at  $m_2$ .

This leads to the following refined assertion:

$$\hat{\varphi}: |\text{ack}| + |\text{send}| + \text{at-}\ell_3 + \text{at-}\underline{m_2} = N.$$

This assertion states that the sum  $|\text{ack}| + |\text{send}|$  essentially equals  $N$ , except when control is at  $\ell_3$  or at  $m_2$ , where each of these locations independently contributes a deficit of 1 to the sum. Thus, when control is both at  $\ell_3$  and at  $m_2$ , the sum equals  $N - 2$ .

It is not difficult to check that assertion  $\hat{\varphi}$  is indeed inductive. Consider, for example, the verification condition for  $\ell_2$ :

$$\rho_{\ell_2} \wedge \underbrace{|\text{ack}| + |\text{send}| + \text{at-}\ell_3 + \text{at-}\underline{m_2} = N}_{\hat{\varphi}} \rightarrow \underbrace{|\text{ack}'| + |\text{send}'| + \text{at}'\ell_3 + \text{at}'\underline{m_2} = N}_{\hat{\varphi}'}$$

Since  $\rho_{\ell_2}$  implies  $\neg \text{at-}\ell_3$ ,  $\text{at}'\ell_3$ ,  $\text{at}'\underline{m_2} = \text{at-}\underline{m_2}$ ,  $\text{send}' = \text{send}$ , and  $\text{ack}' = \text{tl}(\text{ack})$ , it follows that  $|\text{ack}| + \text{at-}\ell_3 = |\text{ack}'| + \text{at}'\ell_3$ , and hence the implication is state valid.

For the refined program, the invariance of  $\hat{\varphi}$  implies the invariance of the goal assertion

$$\psi: |\text{send}| \leq N,$$

which states that the size of the buffer of channel *send* never exceeds  $N$ .

We refer to  $at\_l_3$  and  $at\_m_2$ , appearing in  $\hat{\varphi}$  in the above example, as *compensation expressions*, since they may be viewed as compensating for temporary lapses in the invariance of the expression  $|ack| + |send|$ .

We may summarize our experience with the use of compensation expressions by the following strategy:

#### Strategy (compensation expressions)

An invariant assertion for a coarse version of a program may cease to be invariant when the program is refined.

It may be repaired so as to be invariant for the refined version by the addition of appropriate location-dependent compensation expressions.

#### Example (integer square root)

The invariant assertion that we constructed for the coarse program SQUARE-ROOT\* of Fig. 1.16 is

$$\varphi_1 \wedge \varphi_2 \wedge \varphi_3,$$

where

$$\varphi_1: u = 2 \cdot z + 1 \quad \varphi_2: w = (z + 1)^2 \quad \varphi_3: (z + at\_l_1)^2 \leq x.$$

When refining the program into program SQUARE-ROOT-R of Fig. 1.17, we immediately observed that  $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$  was no longer invariant. In the previous treatment of this example, we assumed that  $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$  still holds at  $l_0$ ,  $l_1$ , and  $l_4$ , and used propagation techniques to derive different versions of  $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$  for the locations  $l_2$  and  $l_3$ .

Following our present strategy, we try to correct each of the three conjuncts,  $\varphi_1$ ,  $\varphi_2$  and  $\varphi_3$ , by adding compensation expressions.

The original conjunct

$$\varphi_1: u = 2 \cdot z + 1$$

was derived based on the assumption that  $z$  and  $u$  were modified in the same transition. In the refined version, this is not the case. Variable  $z$  is incremented first, and only later is variable  $u$  incremented. Therefore, when control is at  $l_2$  there is a deficit of 2 in the value of  $u$ , compared to the right-hand side expression  $2 \cdot z + 1$ . This deficit vanishes as soon as we perform the statement at  $l_2$ . We obtain

$$\hat{\varphi}_1: u + 2 \cdot at\_l_2 = 2 \cdot z + 1.$$

Consider the next conjunct

$$\varphi_2: w = (z + 1)^2.$$

Again, the discrepancy between  $w$  and  $(z + 1)^2$  in the refined version arises only because  $z$  is incremented first, while  $w$  is updated only two statements later. This implies that when we are at  $\ell_2$  or at  $\ell_3$ ,  $w$  still equals  $(z_1 + 1)^2$  where  $z_1$  is the value  $z$  had before the last modification of  $z$ , i.e., at  $\ell_1$ . This value is retrievable from the current value of  $z$  by  $z_1 = z - \text{at-}\ell_{2,3}$ . After execution of  $\ell_3$ ,  $w$  is updated to resume its relation with the current version of  $z$ , so  $w = (z + 1)^2$  becomes valid again. We obtain

$$\hat{\varphi}_2: w = (z + 1 - \text{at-}\ell_{2,3})^2.$$

Finally, consider the conjunct

$$\varphi_3: (z + \text{at-}\ell_1)^2 \leq x.$$

In some sense, we may view  $\varphi_3$  as already containing a compensation expression, and therefore

$$\hat{\varphi}_3 = \varphi_3.$$

This is because, in most locations, the meaning of  $\varphi_3$  is  $z^2 \leq x$ . The only exception is  $\ell_1$  where we are immediately after testing  $w \leq x$ , which implies  $(z + 1)^2 \leq x$ , and just before incrementing  $z$  by 1, which restores the validity of the uncompensated invariant  $z^2 \leq x$ .

It is not difficult to check that the combined assertion

$$\tilde{\varphi}: \hat{\varphi}_1 \wedge \hat{\varphi}_2 \wedge \hat{\varphi}_3$$

is inductive, and is also equivalent to the assertion  $\hat{\varphi}$  we obtained earlier by the propagation techniques, because  $1 - \text{at-}\ell_{2,3} = \text{at-}\ell_{0,1,4}$  is invariant. ■

In **Problem 1.17**, we ask the reader to prove mutual exclusion of a simple version of a producer-consumer program.

## Virtual Variables

As we observed, one of the main reasons for an assertion ceasing to be invariant when the program is refined is that variables that were previously modified together are modified in separate statements in the refined program. The introduction of compensation expressions allows us to refer to previous or future values of variables, rather than to their current values. This idea can be made more explicit by defining the notion of *virtual variables*. Virtual variables are variables that do not appear explicitly in the program but are usually associated with program variables. The association is made explicit by giving each virtual variable a name that resembles the name of the concrete program variable associated with it, e.g.,  $u^*$ ,  $v^*$  being the virtual variables associated with the program variables  $u$ ,  $v$ , respectively.

For each virtual variable we specify a *defining expression* over the program

variables and control predicates. These expressions define the values of the virtual variables over the program states. Typically, at some locations the value of a virtual variable coincides with the value of the actual associated program variable; at other locations it represents either the past value or the future value of the associated actual variable.

### Example (integer square root)

Let us reconsider program SQUARE-ROOT-R of Fig. 1.17.

For the coarse version in Fig. 1.16, we obtained a satisfactory inductive assertion,

$$\varphi: u = 2 \cdot z + 1 \wedge w = (z + 1)^2 \wedge (z + at_{-\ell_1})^2 \leq x,$$

based on the fact that  $z$ ,  $u$ , and  $w$  are modified together. Unfortunately, in the refined version, these variables are modified by different statements. Previously, we used compensation expressions to overcome this problem.

An alternative solution to this problem is to forsake the real variables and to reason instead in terms of ideal *virtual variables* which are modified in transitions of our choice. For the refined version of the program, we may prefer virtual variables  $z^*$ ,  $u^*$ ,  $w^*$ , which, unlike their concrete counterparts  $z$ ,  $u$ , and  $w$ , are not changed by the statements  $\ell_1$  or  $\ell_2$ , but are jointly updated by the statement  $\ell_3$ .

Once we identify the desired behavior of the virtual variables, it is straightforward to define them in terms of the program variables and appropriate location predicates. For the refined version of the program, these definitions are given by

$$\begin{aligned} z^* &= z - at_{-\ell_{2,3}} \\ u^* &= u - 2 \cdot at_{-\ell_3} \\ w^* &= w. \end{aligned}$$

It is not difficult to see that none of  $z^*$ ,  $u^*$ , and  $w^*$  are changed by the statements at  $\ell_1$  and  $\ell_2$ , even though  $z$  and  $u$  are. This is because the changes in  $z$  and  $u$  are compensated for by the control predicates appearing in the defining expressions of the virtual variables, which keep  $z^*$  and  $u^*$  constant over the execution of  $\ell_1$  and  $\ell_2$ .

However, the execution of the statement at  $\ell_3$  preserves  $z$  and  $u$ , and is governed by  $\rho_{\ell_3}$  which implies  $at_{-\ell_3} = 1$ ,  $at'_{-\ell_2} = at'_{-\ell_3} = 0$ , and  $w' = w + u$ . Consequently,  $\ell_3$  modifies the virtual variables as follows:

$$\begin{aligned} (z^*)' &= z - at'_{-\ell_{2,3}} = z = z - at_{-\ell_{2,3}} + 1 = z^* + 1 \\ (u^*)' &= u - 2 \cdot at'_{-\ell_3} = u = u - 2 \cdot at_{-\ell_3} + 2 = u^* + 2 \\ (w^*)' &= w' = w + u = w + u - 2 \cdot at_{-\ell_3} + 2 = w^* + u^* + 2. \end{aligned}$$

The three virtual variables  $z^*$ ,  $u^*$ , and  $w^*$  are found to behave in the refined version completely analogously to the behavior of  $z$ ,  $u$ , and  $w$  in the coarse version of the program. It is therefore not surprising to find that the “virtualized” version of  $\varphi$ ,

$$\varphi^*: \quad u^* = 2 \cdot z^* + 1 \wedge w^* = (z^* + 1)^2 \wedge (z^* + at_{-\ell_1..3})^2 \leq x,$$

is an inductive assertion of the refined version of the program. The extension of the expression  $z + at_{-\ell_1}$  to  $z^* + at_{-\ell_1..3}$  is necessary, since  $z^*$  does not change until  $\ell_3$ .

Note that while establishing the inductiveness of  $\varphi^*$  we repeatedly use the definitions of the virtual variables in terms of the program variables and the control predicates. This is because the transition relation  $\rho_T$  is expressed in terms of these variables and control predicates. ■

### Example (binomial coefficient)

As an additional example using the virtual-variables approach, consider program **BINOM**, presented in Fig. 1.20. This is a correct LCR-program for concurrent computation of the binomial coefficient  $\binom{n}{k}$ .

The computation of the binomial coefficient in this program follows the formula

$$\binom{n}{k} = \frac{n \cdot (n - 1) \cdot \dots \cdot (n - k + 1)}{1 \cdot 2 \cdot \dots \cdot k}.$$

Process  $P_1$  computes the numerator of this formula by successively multiplying into  $b$  the factors  $n, n - 1, \dots, n - k + 1$ . These factors are successively computed in variable  $y_1$ . Process  $P_2$ , responsible for the denominator, successively divides  $b$  by the factors  $1, 2, \dots, k$ , using the integer-division operator  $div$ . These factors are successively computed in variable  $y_2$ .

For the algorithm to be correct, it is necessary that whenever  $div$  is applied it yields no remainder. We rely here on a general property of the integers by which a product of  $m$  consecutive integers is evenly divisible by  $m!$ . Thus,  $b$  should be divided by  $y_2$ , which completes the stage of dividing  $b$  by  $y_2!$ , only when at least  $y_2$  factors have already been multiplied into  $b$  by  $P_1$ . Since  $P_1$  multiplies  $b$  by  $n, n - 1$ , etc., and  $y_1$  is greater than or equal to the value of the next factor to be multiplied, the number of factors that have been multiplied into  $b$  is at least  $n - y_1$ . Therefore,  $y_2$  divides  $b$  as soon as  $y_2 \leq n - y_1$ , or equivalently,  $y_1 + y_2 \leq n$ . This condition, tested at statement  $m_1$ , ensures that  $b$  is divided by  $y_2$  only when it is safe to do so.

The semaphore statements at  $\ell_1$  and  $m_2$  protect the regions  $\ell_{2,3}$  and  $m_{3,4}$  from interference. They guarantee that the value of  $b$  is not modified between its retrieval at  $\ell_2$  and  $m_3$  and its updating at  $\ell_3$  and  $m_4$ .

in  $k, n$  : integer where  $0 \leq k \leq n$   
 local  $y_1, y_2, r$ : integer where  $y_1 = n, y_2 = 1, r = 1$   
 out  $b$  : integer where  $b = 1$

$P_1 :: \left[ \begin{array}{l} \text{local } t_1: \text{integer} \\ \ell_0: \text{while } y_1 > (n - k) \text{ do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{request } r \\ \ell_2: t_1 := b \cdot y_1 \\ \ell_3: b := t_1 \\ \ell_4: \text{release } r \\ \ell_5: y_1 := y_1 - 1 \end{array} \right] \\ \ell_6: \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} \text{local } t_2: \text{integer} \\ m_0: \text{while } y_2 \leq k \text{ do} \\ \quad \left[ \begin{array}{l} m_1: \text{await } y_1 + y_2 \leq n \\ m_2: \text{request } r \\ m_3: t_2 := b \text{ div } y_2 \\ m_4: b := t_2 \\ m_5: \text{release } r \\ m_6: y_2 := y_2 + 1 \end{array} \right] \\ m_7: \end{array} \right]$

Fig. 1.20. Program BINOM (binomial coefficient).

The partial correctness statement for this program is given by the invariance of

$$\psi: \text{at\_}\ell_6 \wedge \text{at\_}m_7 \rightarrow b = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{1 \cdot 2 \cdot \dots \cdot k}$$

which states that, on termination,  $b = \binom{n}{k}$ .

An obvious bottom-up invariant is derived from the standard use of semaphores. It specifies the mutual-exclusion property by claiming the invariance of

$$\chi_1: \neg(\text{at\_}\ell_{2..4} \wedge \text{at\_}m_{3..5})$$

However, for the more intricate invariant concerning the computed value of  $b$ ,

we suggest first analyzing a coarser version whose refinement may yield program BINOM of Fig. 1.20. Program BINOM-C of Fig. 1.21 provides such a coarser version. Note that turning the loop's entire body into a single atomic statement obviates the need for semaphores.

```

in      k, n : integer where  $0 \leq k \leq n$ 
local   y1, y2: integer where  $y_1 = n$ ,  $y_2 = 1$ 
out     b      : integer where  $b = 1$ 

P1 ::  $\left[ \begin{array}{l} \ell_0: \text{while } y_1 > (n - k) \text{ do} \\ \quad \ell_1: (y_1, b) := (y_1 - 1, b \cdot y_1) \\ \ell_6: \end{array} \right]$ 
||

P2 ::  $\left[ \begin{array}{l} m_0: \text{while } y_2 \leq k \text{ do} \\ \quad m_1: \left\langle \begin{array}{l} \text{when } y_1 + y_2 \leq n \text{ do} \\ \quad (y_2, b) := (y_2 + 1, b \text{ div } y_2) \end{array} \right\rangle \\ m_7: \end{array} \right]$ 

```

Fig. 1.21. Program BINOM-C (binomial coefficient) — coarse version.

The partial correctness statement of program BINOM-C (Fig. 1.21) is still given by assertion  $\psi$ .

- *Invariance of  $\psi$  (coarse program)*

To find an inductive strengthening of  $\psi$ , we will try to generalize it. We observe that  $y_1$  measures progress in the computation of the numerator of  $b$ , while  $y_2$  measures progress in the computation of the denominator of  $b$ . This leads to the candidate supplementing assertion

$$\varphi_0: b = \frac{n \cdot (n - 1) \cdot \dots \cdot (y_1 + 1)}{1 \cdot 2 \cdot \dots \cdot (y_2 - 1)}.$$

- Establishing  $\varphi_0 \rightarrow \psi$ :

The assertion  $\varphi_0$  will imply  $\psi$  only if, in addition, we have

$$at_{-\ell_6} \wedge at_{-m_7} \rightarrow y_1 = n - k \wedge y_2 = k + 1.$$

To guarantee this, we propose the two additional assertions

$$\varphi_1: at_{-\ell_6} \rightarrow y_1 = n - k$$

$$\varphi_2: at\_m7 \rightarrow y_2 = k + 1.$$

Later, we will verify that  $\varphi_1$  and  $\varphi_2$  are invariants of the program. Let us concentrate on showing that  $\varphi_0$  is inductive.

- Establishing  $\Theta \rightarrow \varphi_0$ :

Initially,  $b = 1$ ,  $y_1 = n$ , and  $y_2 = 1$ . Interpreting a product of an empty sequence to be 1,  $\varphi_0$  initially holds.

- Establishing  $\{\varphi_0\} T \{\varphi_0\}$ :

There are only two transitions,  $\ell_1$  and  $m_1$ , that modify any of the data variables on which  $\varphi_0$  depends. Consequently, we will only consider the verification conditions for these two.

Consider the verification condition for  $\ell_1$ . Here we have to show

$$\rho_{\ell_1} \wedge b = \underbrace{\frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y_2-1)}}_{\varphi_0} \rightarrow b' = \underbrace{\frac{n \cdot (n-1) \cdots (y'_1+1)}{1 \cdot 2 \cdots (y_2-1)}}_{\varphi'_0}.$$

Since  $\rho_{\ell_1}$  implies  $y'_1 = y_1 - 1$  and  $b' = b \cdot y_1$ , we can multiply both sides of the equality  $\varphi_0$  appearing on the left-hand side of the implication by  $y_1$  to get

$$b \cdot y_1 = \frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y_2-1)} \cdot y_1 = \frac{n \cdot (n-1) \cdots (y_1-1+1)}{1 \cdot 2 \cdots (y_2-1)}$$

which is equivalent to

$$b' = \frac{n \cdot (n-1) \cdots (y'_1+1)}{1 \cdot 2 \cdots (y_2-1)}.$$

Next, consider the verification condition for  $m_1$

$$\rho_{m_1} \wedge b = \underbrace{\frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y_2-1)}}_{\varphi_0} \rightarrow b' = \underbrace{\frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y'_2-1)}}_{\varphi'_0}.$$

The transition relation  $\rho_{m_1}$  implies  $y_1 + y_2 \leq n$ ,  $y'_2 = y_2 + 1$ , and  $b' = b \text{ div } y_2$ . From  $\varphi_0$  and the fact that  $y_1 + y_2 \leq n$ , it follows that  $b$  is evenly divisible by  $y_2$ . This is due to the fact that the numerator in  $b$ ,  $n \cdot (n-1) \cdots (y_1+1)$ , contains the product of  $n - y_1 \geq y_2$  consecutive integers, and is therefore evenly divisible by  $y_2!$ .

Hence, we may rewrite  $b' = b \text{ div } y_2$  as  $b' = b/y_2$ . Dividing by  $y_2$  both sides of the equality  $\varphi_0$ , appearing on the left-hand side of the implication, we obtain

$$b/y_2 = \frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots y_2} = \frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y_2+1-1)}$$

which is equivalent to

$$b' = \frac{n \cdot (n-1) \cdots (y_1+1)}{1 \cdot 2 \cdots (y'_2 - 1)}.$$

This establishes the fact that  $\varphi_0$  is an invariant of program BINOM-C presented in Fig. 1.21.

To conclude from it the invariance of  $\psi$ , which was our original goal, we still need to establish the invariance of  $\varphi_1$  and  $\varphi_2$ , given by

$$\varphi_1: \text{at\_}\ell_6 \rightarrow y_1 = n - k$$

$$\varphi_2: \text{at\_}\ell_7 \rightarrow y_2 = k + 1.$$

- Establishing the invariance of  $\varphi_1$ :

To establish the invariance of  $\varphi_1$ , which is not inductive, we apply propagation as follows. Propagating from  $\ell_6$  to  $\ell_0$  via  $\ell_0^F$ , we obtain

$$\text{at\_}\ell_0 \rightarrow (y_1 \leq n - k \rightarrow y_1 = n - k),$$

which simplifies to

$$\text{at\_}\ell_0 \rightarrow y_1 \geq n - k.$$

Propagating from  $\ell_0$  to  $\ell_1$  via the transition  $\ell_1$ , we obtain

$$\text{at\_}\ell_1 \rightarrow y_1 \geq n - k + 1.$$

Combining these supplementing assertions with  $\varphi_1$ , we obtain the assertion

$$\widehat{\varphi}_1: (\text{at\_}\ell_6 \rightarrow y_1 = n - k) \wedge (y_1 \geq n - k + \text{at\_}\ell_1),$$

which can be shown to be inductive. Clearly,  $\widehat{\varphi}_1$  strengthens  $\varphi_1$  which implies the invariance of  $\varphi_1$ .

- Establishing the invariance of  $\varphi_2$ :

In a completely symmetric way, we derive the inductive assertion

$$\widehat{\varphi}_2: (\text{at\_}\ell_7 \rightarrow y_2 = k + 1) \wedge (y_2 \leq k + 1 - \text{at\_}\ell_1),$$

which establishes the invariance of  $\varphi_2$ .

Consider now the refined program BINOM (Fig. 1.20).

- *Invariance of  $\psi$  (refined program)*

Motivated by the intended role of virtual variables, we would like to introduce virtual variables  $y_1^*$ ,  $y_2^*$ , and  $b^*$ , such that  $y_1^*$  and  $b^*$  are jointly modified by a single transition of  $P_1$ , and  $y_2^*$  and  $b^*$  are jointly modified by a single transition of  $P_2$ .

Considering the situation in  $P_1$ , there seem to be two options. We can either postpone the modification of  $b^*$  from  $\ell_3$  (where  $b$  is modified) to  $\ell_5$ , which modifies

$y_1$ , or shift the modification of  $y_1^*$  from  $\ell_5$  (where  $y_1$  is modified) to the earlier statement  $\ell_3$ . In fact, the two options are not symmetric. Since  $b$  is modified by both processes, it is better to leave the modifications of  $b^*$  at the statements that modify  $b$  and to shift the modifications of  $y_1^*$  and  $y_2^*$  to the same statements.

Consequently, we only define virtual versions of  $y_1$  and  $y_2$ :

$$y_1^* = y_1 - \text{at\_}\ell_{4,5}$$

$$y_2^* = y_2 + \text{at\_}\ell_{5,6}$$

These definitions ensure that while  $y_1$  is decremented by  $\ell_5$ ,  $y_1^*$  is decremented by  $\ell_3$  and is not modified by  $\ell_5$ . Similarly,  $y_2^*$  is modified by  $m_4$ , which increments it by 1.

With these definitions we will establish the invariance of the assertion

$$\varphi_0^*: b = \frac{n \cdot (n-1) \cdots (y_1^*+1)}{1 \cdot 2 \cdots (y_2^*-1)}$$

which can be viewed as the virtual version of the inductive assertion  $\varphi_0$  used for the coarse program.

- $\varphi_0^*$  is inductive:

When checking  $\varphi_0^*$  for inductiveness, we may try to draw analogies with the inductiveness of  $\varphi_0$ .

We discover that a reasonable condition for inductiveness is that the execution of the statement  $\ell_3$  induces the transformation

$$b' = b \cdot y_1^* \quad \text{and} \quad (y_1^*)' = y_1^* - 1.$$

The second equality is obvious from the definition of  $y_1^*$  and the fact that  $\ell_3$  does not modify  $y_1$ . However, to ensure the first equality, we must have

$$\varphi_3: \text{at\_}\ell_3 \rightarrow t_1 = b \cdot y_1^*.$$

A similar consideration leads to the following sufficient conditions for the preservation of  $\varphi_0^*$  over the statement  $m_4$ :

$$\varphi_4: \text{at\_}\ell_3 \rightarrow t_2 = b \text{ div } y_2^*$$

$$\varphi_5: \text{at\_}\ell_3 \rightarrow y_1^* + y_2^* \leq n.$$

Let us therefore check the inductiveness of these three assertions.

- $\varphi_3$  and  $\varphi_4$  are inductive:

The two assertions, being location specific, obviously hold initially, since  $\Theta$  implies

$$\text{at\_}\ell_3 = \text{at\_}\ell_4 = F.$$

Only three transitions may potentially endanger the validity of  $\varphi_3$ . They are  $\ell_2$ ,  $\ell_3$ , and  $m_4$ .

Transition  $\ell_2$  establishes

$$t'_1 = b \cdot y_1 = b \cdot y_1^*,$$

which implies  $\varphi'_3$ . Obviously, taking transition  $\ell_3$  falsifies  $at'_\ell \ell_3$ , which implies  $\varphi'_3$ .

Transition  $m_4$  is more interesting. If it were enabled while  $at_\ell \ell_3$  holds, it would indeed have violated  $\varphi_3$ . Fortunately, due to the previously established mutual-exclusion property  $\chi_1$ :  $\neg(at_\ell \ell_{2..4} \wedge at_\ell m_{3..5})$ , when  $m_4$  is enabled,  $at_\ell \ell_3 = at'_\ell \ell_3 = \text{f}$ . This shows that  $\varphi_3$  is preserved under  $m_4$ .

In a similar way, we can show, using mutual exclusion, that  $\varphi_4$  is inductive.

The assertion  $\varphi_5$  has to be strengthened to become inductive. This is because the main event that guarantees the relation  $y_1^* + y_2^* \leq n$  is a successful execution of the *await* statement at  $m_1$ . Consequently, this information is guaranteed at  $m_2$ , but  $\varphi_5$  requires it at  $m_4$ . To guarantee continuous maintenance of this relation, we suggest the assertion

$$\widehat{\varphi}_5: at\_m_{2..4} \rightarrow y_1^* + y_2^* \leq n.$$

- $\widehat{\varphi}_5$  is inductive:

To show that  $\widehat{\varphi}_5$  is inductive, we have to consider only a few transitions. For transition  $m_1$ ,  $\rho_{m_1}$  implies  $y_1 + y_2 \leq n$  and  $at'_\ell m_2$ . Since at  $m_2$  we have  $y_2^* = y_2$ , and by the definition of  $y_1^*$  we always have  $y_1^* \leq y_1$ , it follows that  $y_1^* + y_2^* \leq n$ . Transitions  $m_2, m_3$  do not modify the value of  $y_1^* + y_2^*$ , and therefore preserve  $\widehat{\varphi}_5$ . The only transition of  $P_1$  that modifies  $y_1^*$  is  $\ell_3$ , and it yields  $(y_1^*)' = y_1^* - 1$ . Consequently, all transitions of  $P_1$  preserve  $\widehat{\varphi}_5$ .

This shows that the conjunction

$$\varphi_0^* \wedge \varphi_3 \wedge \varphi_4 \wedge \widehat{\varphi}_5$$

is inductive, proving the invariance of  $\varphi_0^*$ .

- Establishing  $\psi$ :

In the analysis of BINOM-C, we also needed, in addition to  $\varphi_0$ , the invariance of

$$\varphi_1: at_\ell \ell_6 \rightarrow y_1 = n - k,$$

$$\varphi_2: at_\ell m_7 \rightarrow y_2 = k + 1,$$

to complete the proof of the invariance of  $\psi$ . Since these two assertions were not inductive by themselves, we strengthened them into

$$\widehat{\varphi}_1: (at_\ell \ell_6 \rightarrow y_1 = n - k) \wedge (y_1 \geq n - k + at_\ell \ell_1),$$

$$\widehat{\varphi}_2: (at_\ell m_7 \rightarrow y_2 = k + 1) \wedge (y_2 \leq k + 1 - at_\ell m_1).$$

These assertions are still invariant over the refined program, but they are no longer inductive over it. The best approach is to start again with the goals  $\varphi_1$  and  $\varphi_2$  and apply propagation over the refined version of the program. This leads to the following inductive assertions:

$$\begin{aligned}\tilde{\varphi}_1: \quad & (at\_l_6 \rightarrow y_1 = n - k) \wedge (y_1 \geq n - k + at\_{l1..5}) \\ \tilde{\varphi}_2: \quad & (at\_m_7 \rightarrow y_2 = k + 1) \wedge (y_2 \leq k + 1 - at\_{m1..6}).\end{aligned}$$

To conclude, we argue that the conjunction of  $\varphi_0^*$ ,  $\tilde{\varphi}_1$ , and  $\tilde{\varphi}_2$  implies

$$\psi: \quad at\_l_6 \wedge at\_m_7 \rightarrow b = \frac{n \cdot (n-1) \cdots (n-k+1)}{1 \cdot 2 \cdots k}.$$

To see this, we use the fact that  $at\_l_6$  implies both  $y_1 = n - k$  (due to  $\tilde{\varphi}_1$ ) and  $y_1^* = y_1$  (by the definition of  $y_1^*$ ). Similarly,  $at\_m_7$  implies both  $y_2 = k + 1$  and  $y_2^* = y_2$ . ■

## Auxiliary Program Variables

In the presentation of virtual variables, we identified them as *ideal variables*, that is, variables that do not appear in the program but simplify the proof. Virtual variables are introduced into a program by associating with each variable a defining expression in terms of the program variables.

A different way to introduce ideal variables is by adding variables and assignments to the actual program. We refer to such variables that are added to the program text as *auxiliary program variables*, and to the transformation effecting this addition as *augmentation* (by auxiliary variables).

It is important to realize that the auxiliary program variables are introduced only to aid in the verification. They are not meant to be implemented in the program. Therefore, we do not have to worry about coarse granularity or violations of the LCR-requirements caused by their introduction.

## Augmentation of Fair Transition Systems

We first study the augmentation transformation in the more abstract setting of fair transition systems.

Let  $P: (V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C})$  be a fair transition system. The fair transition system  $\widehat{P}: (\widehat{V}, \widehat{\Theta}, \widehat{\mathcal{T}}, \widehat{\mathcal{J}}, \widehat{\mathcal{C}})$  is called an *augmentation* of  $P$ , if there exists a set of variables  $U$  (the auxiliary variables),  $U \cap V = \emptyset$ , satisfying the following:

- $\widehat{V}: V \cup U$ .
- $\widehat{\Theta}: \Theta \wedge u_1 = f_1(V) \wedge \cdots \wedge u_k = f_k(V)$ , for some  $u_1, \dots, u_k \in U$ ,  $k \geq 0$ .  
Thus, in addition to constraining the initial values of variables in  $V$  (by way of  $\Theta$ ),  $\widehat{\Theta}$  may further specify initial values for some of the variables in  $U$ . These initial values may depend on the initial values of  $V$  through the functions  $f_1(V), \dots, f_k(V)$ .
- $\widehat{\mathcal{T}}: \{\widehat{\tau} \mid \tau \in \mathcal{T}\}$  — that is,  $\widehat{\mathcal{T}}$  contains a transition  $\widehat{\tau}$  for each  $\tau \in \mathcal{T}$ . The transition relation for  $\widehat{\tau}$ , which we denote by  $\widehat{\rho}_{\tau}$ , must be of the form

$$\widehat{\rho}_\tau: \rho_\tau \wedge u'_1 = g_1(U, V, V') \wedge \cdots \wedge u'_m = g_m(U, V, V'),$$

where  $U = \{u_1, \dots, u_m\}$ .

That is, in addition to the requirement that  $V$  and  $V'$  satisfy  $\rho_\tau$ ,  $\widehat{\rho}_\tau$  specifies new values for all the auxiliary variables. These values may depend on the old values of  $U$  and  $V$  and on the new values of  $V$ .

- $\widehat{\mathcal{J}}: \{\widehat{\tau} \mid \tau \in \mathcal{J}\}$  and  $\widehat{\mathcal{C}}: \{\widehat{\tau} \mid \tau \in \mathcal{C}\}$ .

Thus,  $\widehat{\mathcal{J}}$  and  $\widehat{\mathcal{C}}$  contain precisely the augmented versions of the transitions belonging to  $\mathcal{J}$  and  $\mathcal{C}$ , respectively.

**Example** Consider the fair transition system **SQUARE** given by

$$V: \{u, w: \text{integer}\}$$

$$\Theta: u = 1 \wedge w = 0$$

$$\mathcal{T}: \{\tau_I, \tau\}, \text{ where}$$

$$\rho_\tau: u' = u + 2 \wedge w' = w + u$$

$$\mathcal{J} = \mathcal{C}: \{ \}.$$

The following transition system **SQUARE** is an augmentation of **SQUARE** with one auxiliary variable  $U = \{z\}$ :

$$\widehat{V}: \{u, w, z: \text{integer}\}$$

$$\widehat{\Theta}: u = 1 \wedge w = 0 \wedge z = 0$$

$$\widehat{\mathcal{T}}: \{\widehat{\tau}_I, \widehat{\tau}\} \text{ with the transition relation}$$

$$\widehat{\rho}_\tau: u' = u + 2 \wedge w' = w + u \wedge z' = z + 1$$

$$\widehat{\mathcal{J}} = \widehat{\mathcal{C}}: \{ \}.$$

As all the functions considered in this book are assumed to be total, it is not difficult to see that if assertion  $\Theta$  is satisfiable then so is  $\widehat{\Theta}$ , and if transition  $\tau$  is enabled on state  $s$  then so is  $\widehat{\tau}$ . System  $P$ , being a fair transition system, satisfies the requirements that  $\Theta$  is satisfiable and that every state has some transition enabled on it. It follows that  $\widehat{P}$  also satisfies these two requirements and is therefore a legitimate fair transition system.

### Relations between a System and its Augmented Version

Let us represent the states of a fair transition system  $P$  as  $s: \langle a_1, \dots, a_n \rangle$ , implying that  $s[y_i] = a_i$  for each  $y_i \in V$ , and the states of  $\widehat{P}$  as  $\widehat{s}: \langle a_1, \dots, a_n; b_1, \dots, b_m \rangle$  implying that  $\widehat{s}[y_i] = a_i$  for each  $y_i \in V$  and  $\widehat{s}[u_j] = b_j$  for each  $u_j \in U$ .

In the definition of the augmenting transformation, we took special care to ensure that whatever system  $P$  can do with its variables  $V$ , system  $\widehat{P}$  can mimic while assigning appropriate values to the auxiliary variables  $U$ . Thus, if the state

$s_0: \langle a_1^0, \dots, a_n^0 \rangle$  is an acceptable initial state for  $P$ , i.e.,  $s_0 \models \Theta$ , there always exist values  $b_1^0, \dots, b_m^0$  such that  $\hat{s}_0: \langle a_1^0, \dots, a_n^0; b_1^0, \dots, b_m^0 \rangle$  is initial for  $\hat{P}$ , i.e.,  $\hat{s}_0 \models \hat{\Theta}$ .

In a similar way, if the state  $s_2: \langle a_1^2, \dots, a_n^2 \rangle$  is a  $\tau$ -successor of the state  $s_1: \langle a_1^1, \dots, a_n^1 \rangle$  and  $b_1^1, \dots, b_m^1$  is an arbitrary (type-consistent) list of values for  $U$  in  $\hat{s}_1$ , there always exist values  $b_1^2, \dots, b_m^2$  such that

$\hat{s}_2: \langle a_1^2, \dots, a_n^2; b_1^2, \dots, b_m^2 \rangle$  is a  $\hat{\tau}$ -successor of  $\hat{s}_1: \langle a_1^1, \dots, a_n^1; b_1^1, \dots, b_m^1 \rangle$ .

This leads to the following relations between computations of  $P$  and computations of  $\hat{P}$ :

#### Claim (relation between computations)

- Every computation of  $\hat{P}$  is a computation of  $P$ .
- If  $\sigma: s_0, s_1, \dots$  is a computation of  $P$ , there exists a computation of  $\hat{P}$   $\hat{\sigma}: \hat{s}_0, \hat{s}_1, \dots$ , that agrees with  $\sigma$  on the values of  $V$ , i.e.,  $\hat{s}_i[V] = s_i[V]$  for all  $i = 0, 1, \dots$ .

**Example** Consider the following computation of transition system SQUARE:

$\sigma: \langle u: 1, w: 0 \rangle, \langle u: 3, w: 1 \rangle, \langle u: 5, w: 4 \rangle, \langle u: 7, w: 9 \rangle, \dots$

The following computation of  $\widehat{\text{SQUARE}}$  agrees with  $\sigma$  on the values of  $u$  and  $w$ , while assigning values also to the auxiliary variable  $z$ :

$\hat{\sigma}: \langle u: 1, w: 0, z: 0 \rangle, \langle u: 3, w: 1, z: 1 \rangle, \langle u: 5, w: 4, z: 2 \rangle, \langle u: 7, w: 9, z: 3 \rangle, \dots$  ■

Let  $\psi(V)$  be a temporal formula that is valid over the augmented system  $\hat{P}$ , and such that  $\psi$  does not refer to the auxiliary variables  $U$ . Let  $\sigma$  be an arbitrary computation of  $P$ . By the preceding claim, there exists a computation  $\hat{\sigma}$  of  $\hat{P}$  that agrees with  $\sigma$  on the values of  $V$ . Since  $\psi$  is valid over  $\hat{P}$ , we have  $\hat{\sigma} \models \psi$ . As  $\psi$  depends only on  $V$ , it also follows that  $\sigma \models \psi$ . Since  $\sigma$  is an arbitrary computation of  $P$ , we conclude that  $\psi$  is  $P$ -valid. This is summarized in the following claim:

#### Claim (validity transfer)

Let  $\psi$  be a formula that does not refer to any of the auxiliary variables. Then  $\psi$  is valid over  $P$  iff it is valid over its augmented version  $\hat{P}$ .

This claim suggests a proof method by which we augment  $P$  to obtain  $\hat{P}$  and then prove  $\psi$  over  $\hat{P}$ .

**Example** This method can be applied to establish the invariance over fair transition system SQUARE of the assertion

$\psi: \exists k: w = k^2,$

claiming that the value of  $w$  is a square of an integer.

We consider the augmented system  $\widehat{\text{SQUARE}}$  and establish for it the invariance of the assertion

$$\widehat{\varphi}: \quad u = 2 \cdot z + 1 \wedge w = z^2.$$

It is not difficult to check that  $\widehat{\varphi}$  is inductive.

Since  $\widehat{\varphi}$  implies  $\psi$ , we conclude that  $\psi$  is also an invariant of  $\widehat{\text{SQUARE}}$ . Since  $\psi$  does not depend on  $z$ , we invoke the *validity-transfer* claim and conclude that  $\psi$  is also an invariant of  $\text{SQUARE}$ . ■

## Augmentation of Programs

The preceding discussion introduced augmentation as a transformation applied to fair transition systems. In practice, we are mainly interested in systems derived from programs. It is therefore useful to have a definition of augmentation applied directly to programs.

The rules for augmenting a program with auxiliary program variables are the following:

1. Auxiliary variables may be declared at the beginning of any block, with a new mode **auxiliary**.
2. Assignments to auxiliary variables may be added to the program by grouping them together with existing statements.

The grouping may be done either by explicitly grouping an auxiliary assignment with an existing statement or, in the case that the existing statement is an assignment, by transforming the assignment into a multiple assignment with additional entries for the auxiliary variables.

To illustrate the first case, consider a *send* statement

$$\ell: \alpha \Leftarrow e.$$

Suppose we wish to record in an auxiliary variable  $x^a$  the value that was sent on  $\alpha$ , as soon as it was sent. This can be done by explicitly grouping the *send* statement with an assignment, to obtain

$$\ell: \langle \alpha \Leftarrow e; x^a := e \rangle.$$

Illustrating the second case, to augment the existing assignment

$$\ell: u := u - 1$$

by the auxiliary assignment  $v^a := v^a + 1$ , we may use the multiple assignment

$$\ell: (u, v^a) := (u - 1, v^a + 1).$$

Note, in particular, that auxiliary variables are not allowed either on the right-hand side of assignments to program variables, or within tests. Consequently, they cannot affect the values of program variables or the flow of control

in a computation. This observation justifies the augmentation of a program by auxiliary program variables, by implying that the augmented program behaves in precisely the same way as the unaugmented program, except on the auxiliary variables.

In many cases, the auxiliary variables fulfill a role very similar to that of the virtual variables we discussed above. Both types of variables represent a certain amount of dissatisfaction with the suitability of the “real” program variables for expressing elegant invariants, and suggest that some better behaved “ideal” variables will be more suitable for that purpose. The main difference between the two types of ideal variables is in the way they are introduced.

- Virtual variables are introduced by defining them in terms of existing program variables and control predicates.
- Auxiliary program variables are introduced by augmentation of the program with declarations and assignments for the variables. Usually, the equations that were used as definitions for the virtual variables are proved as invariant assertions for the auxiliary variables.

We will illustrate the close analogy between virtual and auxiliary variables on the two programs we considered before.

### Example (integer square root)

Consider first program SQUARE-ROOT-R of Fig. 1.17. We have proven its partial correctness by defining the virtual variables

$$\begin{aligned} z^* &= z - at\_\ell_{2,3} \\ u^* &= u - 2 \cdot at\_\ell_3 \end{aligned}$$

and showing the inductiveness of the assertion

$$\varphi^*: \quad u^* = 2 \cdot z^* + 1 \wedge w = (z^* + 1)^2 \wedge (z^* + at\_\ell_{1..3})^2 \leq x.$$

An alternative approach is to introduce auxiliary program variables  $z^a$  and  $u^a$ . The correspondingly augmented program SQUARE-ROOT-A is presented in Fig. 1.22.

In this program,  $z^a$  and  $u^a$  are no longer virtual variables, but rather variables that are explicitly initialized and manipulated by the program. Consequently, we formulate the following assertions, whose invariance we intend to prove:

$$\begin{aligned} \varphi_1: \quad z^a &= z - at\_\ell_{2,3} \\ \varphi_2: \quad u^a &= u - 2 \cdot at\_\ell_3 \\ \varphi^a: \quad u^a &= 2 \cdot z^a + 1 \wedge w = (z^a + 1)^2 \wedge (z^a + at\_\ell_{1..3})^2 \leq x. \end{aligned}$$

Note that these three assertions are identical to the definitions of  $z^*$  and  $u^*$ , and to the assertion  $\varphi^*$ , respectively, once we replace  $z^*$  and  $u^*$  by  $z^a$  and  $u^a$ , respectively.

```

in      x      : integer where  $x \geq 0$ 
local   u, w   : integer where  $u = 1, w = 1$ 
auxiliary  $z^a, u^a$ : integer where  $z^a = 0, u^a = 1$ 
out     z      : integer where  $z = 0$ 

 $\ell_0$ : while  $w \leq x$  do  $\left[ \begin{array}{l} \ell_1: z := z + 1 \\ \ell_2: u := u + 2 \\ \ell_3: (z^a, u^a, w) := (z^a + 1, u^a + 2, w + u) \end{array} \right]$ 
 $\ell_4$ :

```

Fig. 1.22. Program SQUARE-ROOT-A — augmented.

To prove partial correctness of the program of Fig. 1.22, i.e., the invariance of the assertion

$$\psi: at_{\ell_4} \rightarrow z^2 \leq x < (z + 1)^2,$$

it is sufficient to prove the invariance of the assertions  $\varphi_1$ ,  $\varphi_2$ , and  $\varphi^a$ . Once these assertions are proven invariant, the invariance of  $\psi$  can be proven. We illustrate only the proof for  $\varphi_2$ .

To prove that

$$\varphi_2: u^a = u - 2 \cdot at_{\ell_3}$$

is inductive, we observe that initially  $u^a = u = 1$  and  $at_{\ell_3} = 0$ . The two transitions that have to be checked are  $\ell_2$  and  $\ell_3$ .

The statement  $\ell_2$  modifies  $u$  and  $at_{\ell_3}$ , but not  $u^a$ . Furthermore,  $\rho_{\ell_2}$  implies

$$at_{\ell_3} = 0 \quad at'_{\ell_3} = 1 \quad u' = u + 2,$$

so that

$$\rho_{\ell_2} \wedge \underbrace{u^a = u - 2 \cdot at_{\ell_3}}_{\varphi_2} \rightarrow \underbrace{u^a = u' - 2 \cdot at'_{\ell_3}}_{\varphi'_2},$$

since

$$u' - 2 \cdot at'_{\ell_3} = (u + 2) - 2 = u = u - 2 \cdot at_{\ell_3} = u^a.$$

The statement  $\ell_3$  modifies  $u^a$  and  $at_{\ell_3}$ , but not  $u$ . Furthermore,  $\rho_{\ell_3}$  implies

$$at_{\ell_3} = 1 \quad at'_{\ell_3} = 0 \quad (u^a)' = u^a + 2,$$

so that

$$\rho_{\ell_3} \wedge \underbrace{u^a = u - 2 \cdot at_{-\ell_3}}_{\varphi_2} \rightarrow \underbrace{(u^a)' = u - 2 \cdot at'_{-\ell_3}}_{\varphi'_2},$$

since

$$u - 2 \cdot at'_{-\ell_3} = u = u - 2 \cdot at_{-\ell_3} + 2 = u^a + 2 = (u^a)'. \blacksquare$$

### Example (binomial coefficient)

Consider program BINOM (Fig. 1.20). Again, we can introduce the auxiliary variables  $y_1^a$  and  $y_2^a$  instead of the virtual variables  $y_1^*$  and  $y_2^*$ , as we did before. The augmented program BINOM-A is presented in Fig. 1.23. We have added declarations for  $y_1^a$ ,  $y_2^a$  and changed the assignments at  $\ell_3$  and  $m_4$ , by adding assignments to  $y_1^a$  and  $y_2^a$ .

In proving the partial correctness of this program, it is now necessary to verify the invariance of the following assertions (used previously as definitions):

$$\varphi_1: y_1^a = y_1 - at_{-\ell_{4,5}}$$

$$\varphi_2: y_2^a = y_2 + at_{-m_{5,6}}. \blacksquare$$

## Are Auxiliary Variables Essential?

Some of the early investigations of proof methods for concurrent programs restricted themselves to assertions that are conjunctions of *local assertions*. An assertion  $\varphi$  is called *local to process*  $P_i$  if it only refers to data variables that are referenced in  $P_i$  and to the control variable  $\pi$  through expressions  $at_{-\ell}$ , where  $\ell$  is a location within  $P_i$ . An assertion which is a conjunction of local assertions is called *conjunctively local*.

For example, the most general conjunctively local assertion for program DOUBLE, presented in Fig. 1.24, has the form

$$\varphi(at_{-\ell_0}, at_{-\ell_1}, x) \wedge \psi(at_{-m_0}, at_{-m_1}, x).$$

Working under this restricted type of assertions, it can be shown that

any proof method that only uses conjunctively local assertions would be incomplete without the license to add auxiliary variables.

In **Problem 1.18**, the reader is requested to show that the invariance of the assertion  $at_{-\ell_1} \wedge at_{-m_1} \rightarrow x = 2$ , which is an invariant of program DOUBLE of Fig. 1.24, cannot be proven by using conjunctively local assertions alone.

In the approach taken in this book, no restrictions of locality are imposed and assertions may freely reference all control components of the state. In this framework, auxiliary variables are a convenience rather than a necessity.

in             $k, n$         : integer where  $0 \leq k \leq n$   
 local         $y_1, y_2, r$ : integer where  $y_1 = n, y_2 = 1, r = 1$   
 auxiliary  $y_1^a, y_2^a$  : integer where  $y_1^a = n, y_2^a = 1$   
 out           $b$             : integer where  $b = 1$

$P_1 ::$  
$$\left[ \begin{array}{l} \text{local } t_1: \text{integer} \\ \ell_0: \text{while } y_1 > (n - k) \text{ do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{request } r \\ \ell_2: t_1 := b \cdot y_1 \\ \ell_3: (b, y_1^a) := (t_1, y_1^a - 1) \\ \ell_4: \text{release } r \\ \ell_5: y_1 := y_1 - 1 \end{array} \right] \\ \ell_6: \end{array} \right]$$
  
 $\parallel$   
 $P_2 ::$  
$$\left[ \begin{array}{l} \text{local } t_2: \text{integer} \\ m_0: \text{while } y_2 \leq k \text{ do} \\ \quad \left[ \begin{array}{l} m_1: \text{await } y_1 + y_2 \leq n \\ m_2: \text{request } r \\ m_3: t_2 := b \text{ div } y_2 \\ m_4: (b, y_2^a) := (t_2, y_2^a + 1) \\ m_5: \text{release } r \\ m_6: y_2 := y_2 + 1 \end{array} \right] \\ m_7: \end{array} \right]$$

Fig. 1.23. Program BINOM-A — augmented.

local  $x$ : integer where  $x = 0$

$$\left[ \begin{array}{l} \ell_0: x := x + 1 \\ \ell_1: \end{array} \right] \parallel \left[ \begin{array}{l} m_0: x := x + 1 \\ m_1: \end{array} \right]$$

Fig. 1.24. Program DOUBLE (double incrementation).

In **Problem 1.19**, we ask the reader to verify an invariance property of a given program.

## Problems

**Problem 1.1** (verification condition is  $P$ -valid) page 85

Let  $P$  be a fair transition system,  $\tau \in \mathcal{T}$ , a transition of  $P$ , and  $\varphi$  and  $\psi$  assertions. Assume that every  $\tau$ -successor of every  $P$ -accessible  $\varphi$ -state is a  $\psi$ -state. Show that the verification condition  $\{\varphi\} \tau \{\psi\}$  is  $P$ -state valid.

**Problem 1.2** (converse direction of rule INV-B) page 88

Show that if assertion  $\varphi$  is  $P$ -invariant, then premises B1 and B2 are  $P$ -state valid.

**Problem 1.3** (proving the control invariants) page 110

Establish the  $P$ -invariance of the control invariants CONFLICT, SOMEWHERE, EQUAL, and PARALLEL, introduced in page 109.

**Problem 1.4** (preconditions) page 115

(a) Let  $V = \{x, y\}$ , where  $x$  and  $y$  range over the integers. Suppose  $\tau$  is given by the following transition relation:

$$\rho_\tau: \quad x' = x + 1 \wedge y' = y + 1.$$

Compute:

- (i)  $\text{pre}(\tau, x + y = 5)$
- (ii)  $\text{pre}(\tau, x + y < 3)$
- (iii)  $\text{pre}(\tau, y = 0)$ .

(b) Let  $V = \{x, y, \pi\}$ , where  $x$  and  $y$  range over the integers and  $\pi$  is a control variable. Suppose  $\tau$  is given by the following transition relation:

$$\rho_\tau: \quad \text{move}(\ell_0, \ell_1) \wedge x' = x + 1 \wedge y' = y + 1.$$

Compute

- (i)  $\text{pre}(\tau, x + y = 5)$
- (ii)  $\text{pre}(\tau, x + y < 3)$
- (iii)  $\text{pre}(\tau, y = 0)$ .

(c) Consider program MUX-DEK-A of Fig. 1.29. For the transitions associated with statements  $\ell_2, \ell_3, \ell_4, \ell_5, \ell_6$ , compute  $\text{pre}(\tau, \neg(\text{at\_}\ell_7 \wedge \text{at\_}\ell_7))$ .

**Problem 1.5** (computing the factorial function) page 122

Program FACT of Fig. 1.25 accepts a nonnegative input in variable  $x$  and computes in variable  $z$  the factorial function  $x!$ . Prove partial correctness of program FACT, which can be stated by the  $P$ -invariance of the assertion

$$at\_{\ell_2} \rightarrow z = x!.$$

```

in   x: integer where x ≥ 0
local y: integer where y = 0
out  z: integer where z = 1

ℓ₀: while x ≠ y do
    ℓ₁: (y, z) := (y + 1, (y + 1) · z)
    ℓ₂:

```

Fig. 1.25. Program FACT (computing the factorial function).

**Problem 1.6** (integer division) page 122

Program IDIV of Fig. 1.26 accepts two positive integers in variables  $x$  and  $y$  and places in variable  $z$  their integer quotient  $x \text{ div } y$ , and in variable  $w$  the remainder of their division  $x \text{ mod } y$ . Prove partial correctness of program IDIV, which can be stated by the  $P$ -invariance of the assertion

$$at\_{\ell_5} \rightarrow x = z \cdot y + w \wedge 0 \leq w < y.$$

**Problem 1.7** (greatest common divisor) page 122

Program GCDM of Fig. 1.27 accepts two positive integers in variables  $x_1$  and  $x_2$ . It computes in  $z$  the greatest common divisor ( $gcd$ ) of  $x_1$  and  $x_2$ , and in variables  $w_1$  and  $w_2$  two integers which express  $z$  as a linear combination of the inputs  $x_1$  and  $x_2$ . Prove partial correctness of program GCDM, which can be stated by the  $P$ -invariance of the assertion

$$at\_{\ell_6} \rightarrow z = gcd(x_1, x_2) \wedge z = w_1 \cdot x_1 + w_2 \cdot x_2.$$

The program uses the operation  $\text{div}$  of integer division and the operation  $\text{mod}$  which computes the remainder of an integer division. In your proof you may use the following properties of the  $gcd$  function which hold for all nonzero integers  $m$  and  $n$  (possibly negative):

$$gcd(m, n) = gcd(m - n, n) \quad \text{for every } m \neq n$$

```

in   x, y : integer where x > 0, y > 0
local t    : integer
out  z, w: integer where z = w = 0

 $\ell_0$ :  $t := x$ 
 $\ell_1$ : while  $t > 0$  do
     $\ell_2$ : if  $w + 1 = y$ 
        then  $\ell_3$ :  $(z, w, t) := (z + 1, 0, t - 1)$ 
        else  $\ell_4$ :  $(z, w, t) := (z, w + 1, t - 1)$ 
 $\ell_5$ :

```

Fig. 1.26. Program IDIV (integer division).

```

in   x1, x2          : integer where x1 > 0, x2 > 0
local y1, y2, t1, t2, t3, t4, u: integer
out  z, w          : integer

 $\ell_0$ :  $(y_1, y_2, t_1, t_2, t_3, t_4) := (x_1, x_2, 1, 0, 0, 1)$ 
 $\ell_1$ :  $(y_1, y_2, u) := (y_2 \text{ mod } y_1, y_1, y_2 \text{ div } y_1)$ 
 $\ell_2$ : while  $y_1 \neq 0$  do
     $\left[ \begin{array}{l} \ell_3: (t_1, t_2, t_3, t_4) := (t_2 - u \cdot t_1, t_1, t_4 - u \cdot t_3, t_3) \\ \ell_4: (y_1, y_2, u) := (y_2 \text{ mod } y_1, y_1, y_2 \text{ div } y_1) \end{array} \right]$ 
 $\ell_5$ :  $(z, w_1, w_2) := (y_2, t_2, t_3)$ 
 $\ell_6$ :

```

Fig. 1.27. Program GCDM (greatest common divisor with multipliers).

$$\gcd(m, m) = |m|.$$

**Problem 1.8** (computing the *gcd* and *lcm*) page 122

Program GCDLCM of Fig. 1.28 accepts two positive integers in variables  $x_1$  and  $x_2$ . It computes in variable  $z$  their greatest common divisor and in variable  $w$  their

least common multiple. Prove partial correctness of program GCDLCM, which can be stated by the  $P$ -invariance of the assertion

$$at\_l_7 \rightarrow z = gcd(x_1, x_2) \wedge w = lcm(x_1, x_2).$$

```

in   x1, x2      : integer where x1 > 0, x2 > 0
local y1, y2, y3, y4: integer
out  z, w      : integer

l0: (y1, y2, y3, y4) := (x1, x2, x2, 0)
l1: while y1 ≠ y2 do
    [l2: if y1 > y2 then
        l3: (y1, y4) := (y1 - y2, y3 + y4)
    l4: if y1 < y2 then
        l5: (y2, y3) := (y2 - y1, y3 + y4)]
l6: (z, w) := (y1, y3 + y4)
l7:
```

Fig. 1.28. Program GCDLCM (computing the  $gcd$  and  $lcm$ ).

In your proof you may use the properties of the  $gcd$  function listed in Problem 1.7, and the following property of the  $lcm$  function:

$$lcm(m, n) = \frac{m \cdot n}{gcd(m, n)}.$$

### Problem 1.9 (Dekker's algorithm) page 127

(a) Prove mutual exclusion for program MUX-DEK of Fig. 1.15 by establishing the following invariants

$$\varphi_0: t = 1 \vee t = 2$$

$$\varphi_1: y_1 \leftrightarrow at\_l_{3..5,8..10}$$

$$\varphi_2: y_2 \leftrightarrow at\_m_{3..5,8..10}.$$

and showing that the assertion  $\psi: \neg(at\_l_{8..10} \wedge at\_m_{8..10})$  is invariant.

(b) Prove mutual exclusion for program MUX-DEK-A of Fig. 1.29. That is, show that the assertion  $\neg(at\_l_7 \wedge at\_m_7)$  is invariant.

**local**  $y_1, y_2, t$ : integer where  $y_1 = y_2 = 0, t = 1$

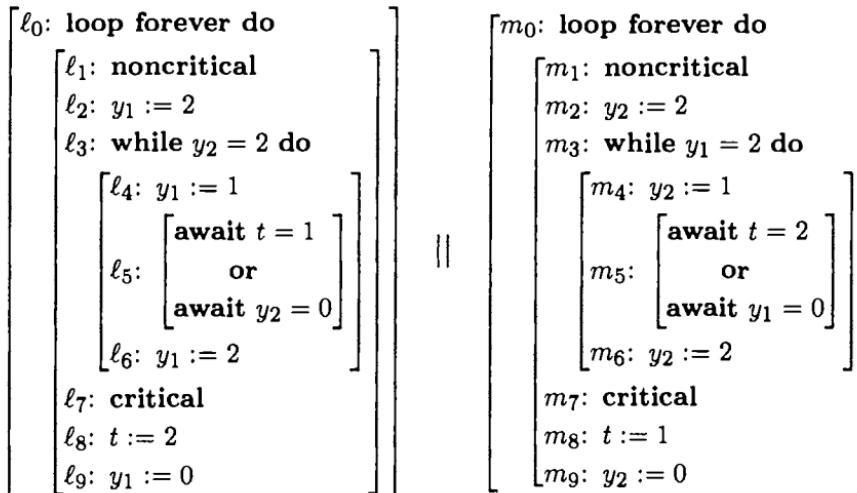


Fig. 1.29. Program MUX-DEK-A (a variant of Dekker's algorithm).

(c) Prove mutual exclusion for program MUX-DEK-B of Fig. 1.30. That is, show that the assertion  $\neg(at\_l_8 \wedge at\_m_8)$  is invariant.

**Problem 1.10** (three values) page 127

Prove mutual exclusion for program MUX-VAL-3 of Fig. 1.31. This program uses the shared integer variables  $y_1$  and  $y_2$ . Obviously, these variables can only assume one of the values  $\{-1, 0, 1\}$ .

Mutual exclusion is stated by the invariance of the assertion

$$\neg(at\_l_4 \wedge at\_m_4).$$

**Problem 1.11** (computing the cube) page 130

Program CUBE of Fig. 1.32 computes the cube of a positive integer input, using only additions. Prove partial correctness of program CUBE, which can be stated by the  $P$ -invariance of the assertion

$$at\_l_7 \rightarrow z = x^3.$$

In your proof, you may use the identity:

$$(n+1)^3 = n^3 + 3(n+1)n + 1.$$

**local**  $y_1, y_2, t$ : **integer** **where**  $y_1 = 0, y_2 = 0, t = 1$

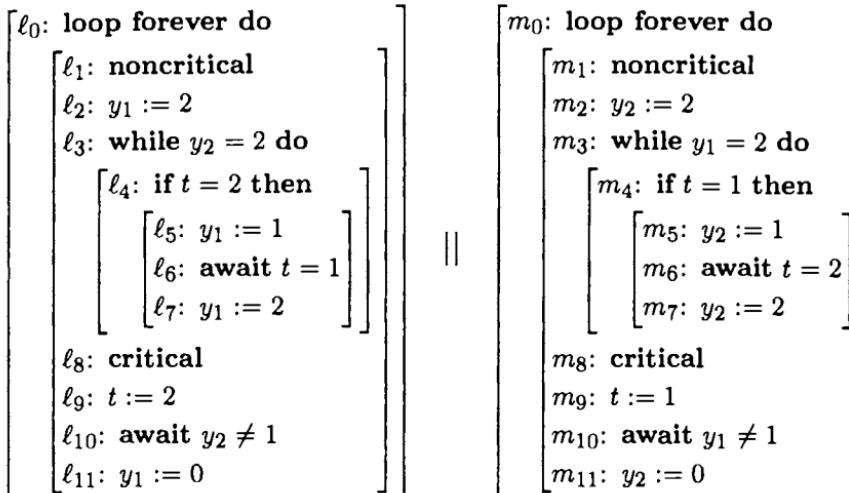


Fig. 1.30. Program MUX-DEK-B.

**local**  $y_1, y_2$ : **integer** **where**  $y_1 = y_2 = 0$

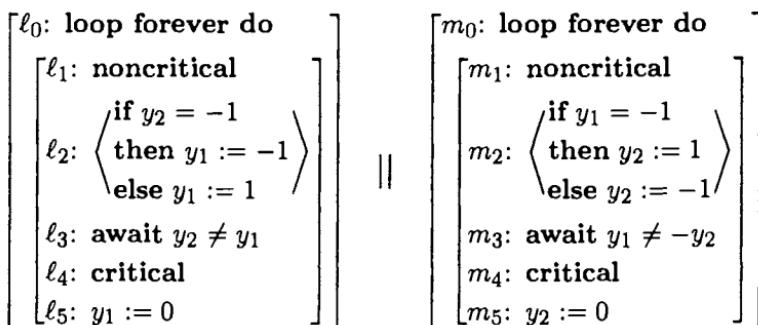


Fig. 1.31. Program MUX-VAL-3.

```

in   x           : integer where  $x > 0$ 
local u, y1, y2, y3 : integer
out  z           : integer

l0: (u, y1, y2, y3) := (0, 0, 0, 0)
l1: while u < x do
    [l2: u := u + 1
     l3: y3 := y3 + y2 + 1
     l4: y1 := y1 + 6
     l5: y2 := y2 + y1]
l6: z := y3
l7:

```

Fig. 1.32. Program CUBE (computing the cubic power).

**Problem 1.12** (bakery algorithms) page 130

- (a) Prove mutual exclusion for program MUX-BAK-A of Fig. 1.33. Note that the two processes are not exactly symmetric due to the difference between statements  $\ell_3^b$  and  $m_3^b$ .

The algorithm is called the *bakery* algorithm, since it is based on the idea that customers, as they enter, pick numbers which form an ascending sequence. Then, a customer with a lower number has higher priority in accessing its critical section. Statements  $\ell_2$  and  $m_2$  ensure that the number assigned to  $y_i$ ,  $i = 1, 2$ , is greater than the current value of  $y_j$ ,  $j \neq i$ .

- (b) Program MUX-BAK-A does not obey the LCR restriction. In particular, statements  $\ell_2$  and  $m_2$  each contain two critical references: to  $y_1$  and to  $y_2$ . To correct this, one may suggest program MUX-BAK-B presented in Fig. 1.34, in which each of the offending statements has been expanded to two assignments. Show that program MUX-BAK-B cannot guarantee mutual exclusion, by presenting a computation in which both processes reside in their critical sections at the same time.

- (c) We observe that mutual exclusion is violated in program MUX-BAK-B only when one of the processes waits for a long time at locations  $\ell_3$  or  $m_3$ . To correct this situation, we propose program MUX-BAK-C of Fig. 1.35. This LCR-program contains two additional *await* statements that ensure that processes do not wait too long at locations  $\ell_3$  or  $m_3$ . Show that program MUX-BAK-C guarantees mutual exclusion.

local  $y_1, y_2$ : integer where  $y_1 = y_2 = 0$

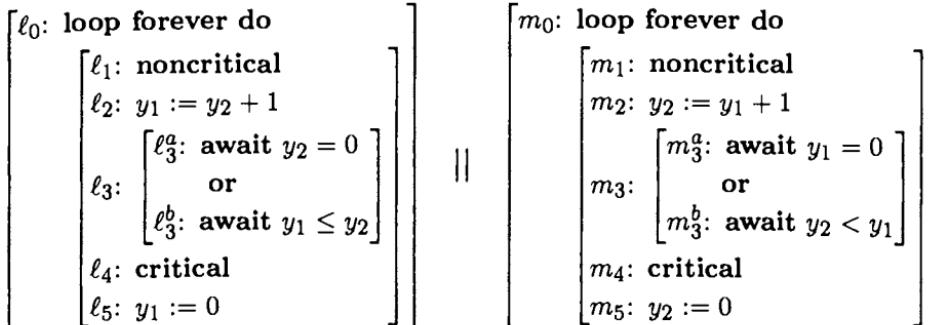


Fig. 1.33. Program MUX-BAK-A.

local  $y_1, y_2, t_1, t_2$ : integer where  $y_1 = y_2 = 0$

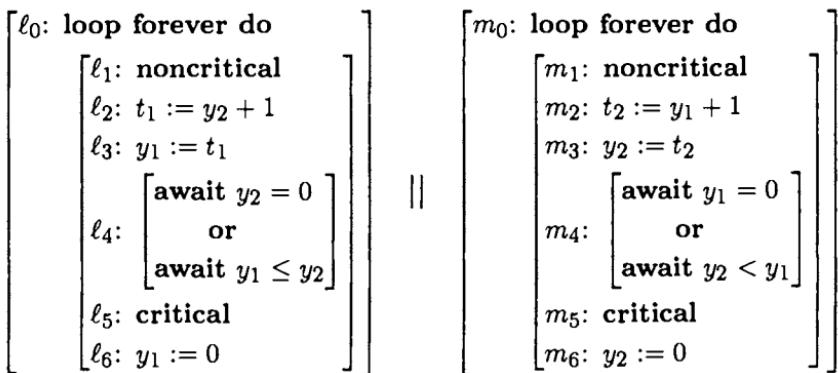


Fig. 1.34. Program MUX-BAK-B.

### Problem 1.13 (*swap* statement) page 130

The **request** statement is not available on all systems. As an alternative some systems offer the following, simpler to implement, instruction that swaps the value of two boolean variables:

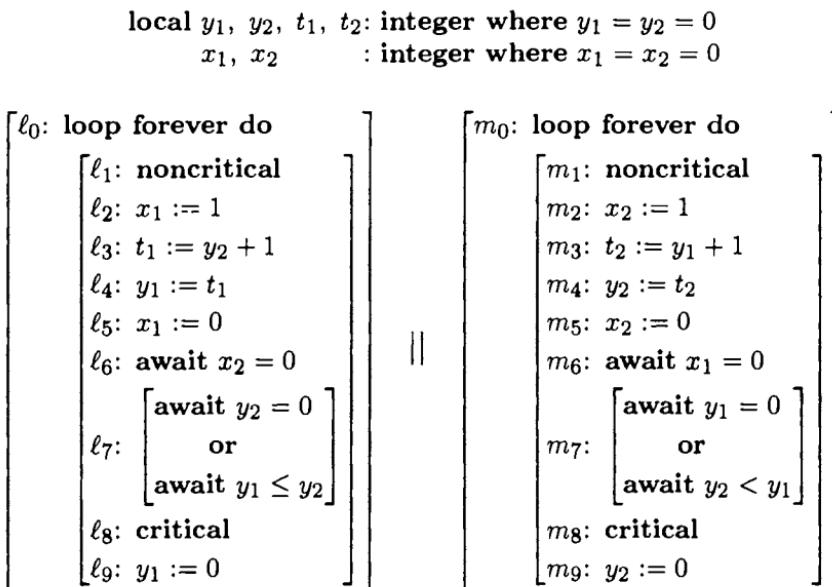


Fig. 1.35. Program MUX-BAK-C.

$\ell$ :  $x ::= y$ ;  $\hat{\ell}$ :

whose transition relation is given by

$$\rho_\ell: move(\ell, \hat{\ell}) \wedge x' = y \wedge y' = x \wedge pres(V - \{x, y\}).$$

(a) Show that  $x ::= y$  is equivalent to the following grouped statement:

$\langle$  **local**  $temp$ : boolean  
 $temp := x$   
 $x := y$   
 $y := temp$   $\rangle$

(b) Show that if we do not group the above statement then the two statements are not equivalent.

(c) Write a program for implementing the semaphore statements **request** and **release** using the **swap** statement.

**Problem 1.14** (mutual exclusion with *swap*) page 130

Consider program MUX-SWAP of Fig. 1.36. Prove mutual exclusion for MUX-SWAP.

Is this an acceptable solution to the mutual-exclusion problem?

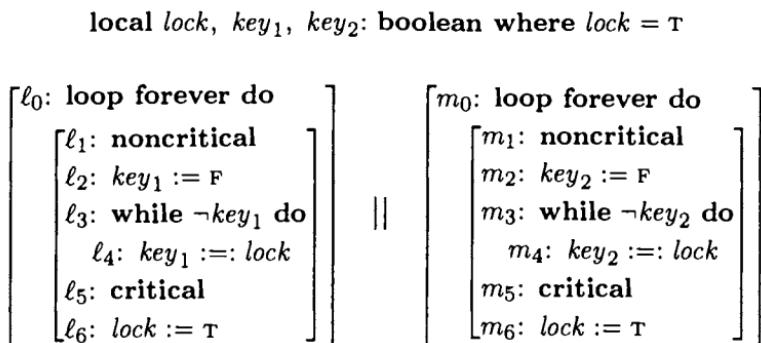


Fig. 1.36. Program MUX-SWAP.

**Problem 1.15** (*test-and-set* statement) page 130

As another alternative to **await**, some systems offer the following function (with side effect) known as **testandset**( $x$ ), which sets the value of the boolean variable  $x$  to true and returns the original value of  $x$  as the value of the statement

$$\ell: y := \text{testandset}(x); \hat{\ell}:$$

whose transition relation is given by

$$\rho_\ell: \text{move}(\ell, \hat{\ell}) \wedge x' = T \wedge y' = x \wedge \text{pres}(V - \{x, y\}).$$

Write a program for implementing the semaphore statements **request** and **release** using the **testandset** function.

**Problem 1.16** (mutual exclusion by *test-and-set*) page 130

Consider program MUX-TESTSET in Fig. 1.37. Statements  $\ell_2$ ,  $m_2$ ,  $\ell_4$ ,  $m_4$ , and part of the statements labeled by  $\ell_3$  and  $m_3$  are missing. Fill in the required statements using **testandset** to ensure mutual exclusion.

**Problem 1.17** (a simple producer-consumer program) page 135

Program PROD-CONS-S of Fig. 1.38 is a simple version of a producer-consumer program, where the producer and consumer communicate by a shared integer variable  $y$ . Prove the property of mutual exclusion for program PROD-CONS-S, which can be expressed by the  $P$ -invariance of the assertion

$$\neg(at - \ell_3 \wedge at - m_2).$$

**Problem 1.18** (auxiliary variables sometimes essential) page 150

(a) Show that the assertion

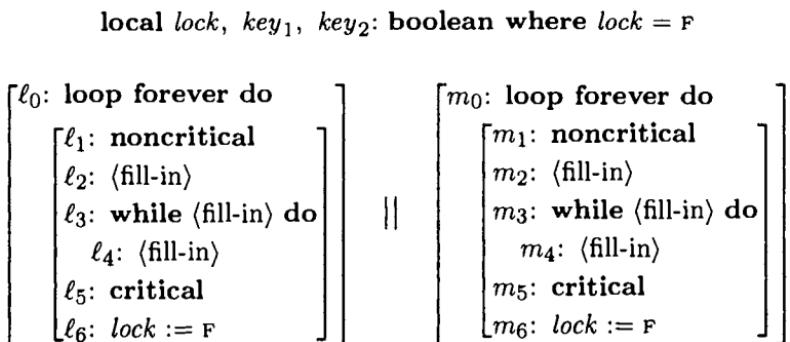


Fig. 1.37. Program MUX-TESTSET.

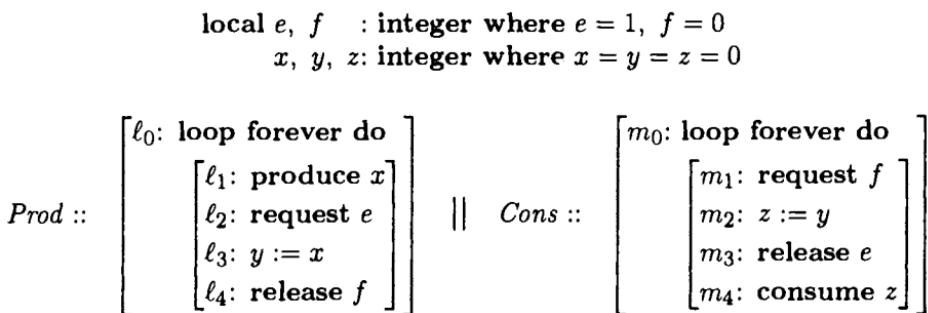


Fig. 1.38. Program PROD-CONS-S (simple version of a producer-consumer program).

$$\varphi: at\_l_1 \wedge at\_m_1 \rightarrow x = 2$$

cannot be proven invariant for program DOUBLE of Fig. 1.24, when we are restricted to the use of auxiliary assertions which are conjunctively local, i.e., have the general form

$$\varphi(at\_l_0, at\_l_1, x) \wedge \psi(at\_m_0, at\_m_1, x).$$

- (b) Augment program DOUBLE by auxiliary variables which enable proving  $\varphi$ , using conjunctively local assertions.
- (c) Consider program DIFF-INC of Fig. 1.39. Show that the assertion

**local**  $x$ : integer where  $x = 0$

$$\left[ \begin{array}{l} \ell_0: x := x + 1 \\ \ell_1: \end{array} \right] \quad || \quad \left[ \begin{array}{l} m_0: x := x + 2 \\ m_1: \end{array} \right]$$

Fig. 1.39. Program DIFF-INC (incrementing by different amounts).

$$\psi: at_{\ell_1} \wedge at_{m_1} \rightarrow x = 3$$

can be proven to be an invariant of program DIFF-INC, using only conjunctively local assertions and without adding auxiliary variables.

### Problem 1.19 (set partitioning) page 152

Consider program EXCH presented in Fig. 1.40. The program accepts as input two sets of natural numbers  $S$  and  $T$ , whose initial values are  $S_0$  and  $T_0$ , respectively.

in  $S_0, T_0$ : set of natural where  $S_0 \neq \emptyset, T_0 \neq \emptyset$   
 out  $S, T$  : set of natural where  $S = S_0, T = T_0$   
**local**  $\alpha, \beta$  : channel of integer

$P_1 ::$ <div style="border: 1px solid black; padding: 10px;"> <p><b>local</b> <math>x, mx</math>: integer</p> <math>\ell_0: x := -1</math> <math>\ell_1: mx := \max(S)</math> <math>\ell_2: \text{while } x &lt; mx \text{ do}</math> <div style="border: 1px solid black; padding: 5px; margin-left: 20px;"> <math>\ell_3: \alpha \Leftarrow mx</math> <math>\ell_4: S := S - \{mx\}</math> <math>\ell_5: \beta \Rightarrow x</math> <math>\ell_6: S := S \cup \{x\}</math> <math>\ell_7: mx := \max(S)</math> </div> <math>\ell_8: \alpha \Leftarrow -1</math> <math>\ell_9:</math> </div>	$\parallel$	$P_2 ::$ <div style="border: 1px solid black; padding: 10px;"> <p><b>local</b> <math>y, mn</math>: integer</p> <math>m_0: \alpha \Rightarrow y</math> <math>m_1: \text{while } y \geq 0 \text{ do}</math> <div style="border: 1px solid black; padding: 5px; margin-left: 20px;"> <math>m_2: T := T \cup \{y\}</math> <math>m_3: mn := \min(T)</math> <math>m_4: \beta \Leftarrow mn</math> <math>m_5: T := T - \{mn\}</math> <math>m_6: \alpha \Rightarrow y</math> </div> <math>m_7:</math> </div>
---	-------------	--

Fig. 1.40. Program EXCH (partitioning two sets).

Process  $P_1$  repeatedly identifies and removes the maximal element in  $S$  and

sends it to  $P_2$  which places it in  $T$ . Symmetrically,  $P_2$  identifies and removes the minimal element in  $T$  and sends it to  $P_1$  which places it in  $S$ . The processes use the operations  $\max(S)$  and  $\min(T)$  which find, respectively, the maximal element in the set  $S$  and the minimal element in the set  $T$ . Show that the following assertion is an invariant of program EXCH

$$\varphi: \text{at\_l}_9 \wedge \text{at\_m}_7 \rightarrow |S| = |S_0| \wedge |T| = |T_0| \wedge S \leq T.$$

This assertion states that, on termination, sets  $S$  and  $T$  have preserved their initial sizes and that every element in  $S$  is smaller than or equal to every element in  $T$ .

## Bibliographic Remarks

The use of invariant assertions for proving properties of programs lies at the very center and sources of program verification. Early sketches of verification methodology appear already in Turing [1950] and Goldstine and von Neumann [1947]. Naur [1966] presents an application of this approach to the verification of a particular algorithm.

A major step in formalizing these ideas was taken in Floyd [1967], suggesting the use of intermediate assertions for proving partial correctness, and the use of well-founded progress measures for establishing termination of sequential programs. See Manna [1969] for elaboration of these techniques and Manna [1974] for a summary of the proof methods for establishing partial and total correctness of sequential programs. Floyd's approach applied to programs presented by flowcharts. Inspired by Floyd's method, Hoare [1969] proposed an axiomatic proof method for verifying sequential programs in a structured syntax-oriented way. This approach received a great deal of attention, and many Hoare-style proof systems dealing with various sequential programming constructs have been proposed since then. We refer the reader to Apt [1981] for a survey of these systems.

Floyd's unstructured method, in which intermediate assertions (invariants) are associated with control points in the program, has been extended to concurrent programs by Ashcroft [1975] (see also Ashcroft and Manna [1971]). The extension associates an intermediate assertion with each tuple of control locations, one belonging to each process. A closely related approach is proposed by Flon and Suzuki [1978] who suggest transforming a concurrent program into an equivalent nondeterministic one, reflecting all possible interleaving sequences, and then applying (an extension of) Floyd's method to the resulting program. At first glance, both approaches seem to be extremely expensive, possibly requiring a number of intermediate assertions which is exponential in the size of the original concurrent program.

The next meaningful step in the verification of concurrent programs was taken by

Owicki and Gries [1976a, 1976b] who proposed a Hoare-style system for concurrent programs. Unlike Hoare's system for sequential programs, the system of Owicki and Gries [1976a] does not discard the intermediate assertions when smaller statements are concatenated to form larger statements. Instead, the whole set of intermediate assertions used for establishing a conclusion about a process  $P_i$  is used to form a *proof outline*. After the local proof for process  $P_i$  is concluded, it is necessary to check for *non-interference*, meaning that each of the assertions used in the local proof of  $P_i$  is preserved by any action of every parallel process  $P_j$ ,  $j \neq i$ . The proof of (relative) completeness of their method depends strongly on the use of *auxiliary variables*.

It is possible to view Owicki and Gries' method as the association of an assertion  $\varphi_\ell$  with each location  $\ell$  in the program, i.e., going back to Floyd's original method. This can be viewed as an improvement on Ashcroft's method which associates an assertion with each tuple of locations.

From here, it is only one step to consider assertions that are not attached to any location but are expected to hold throughout the computation, namely, invariants. Their dependence on the locations can be expressed either via auxiliary variables or by a direct reference to the control through expressions such as  $at\_\ell$ . The utility of this approach depends very much on our success in expressing the dependence on control and other variables in a succinct way. Unattached invariants are used in Keller [1976] and Lamport [1977].

Owicki and Gries' method only applies to systems communicating by shared variables. Its extension to synchronous message-passing systems was done in Apt, Francez, and de Roever [1980] and Levin and Gries [1981].

**The rule:** INV, in the form presented in the text, is taken from Manna and Pnueli [1983c]. However, the underlying principle based on computational induction is a direct consequence of Floyd's method, simplified to the case that the same assertion is attached to all locations, and is also the one used in Keller [1976] and Lamport [1977].

**Construction of invariants:** Most of the heuristics for bottom-up and top-down construction of inductive assertions and invariants are taken from Katz and Manna [1976]. Some of the preceding contributions to this topic are Caplain [1975], Elspas [1974], German [1974], Katz and Manna [1973], Misra [1975], Moriconi [1974], and Wegbreit [1974]. Further techniques for the construction of global invariants were developed by Dershowitz and Manna [1981].

Top-down development of invariants is studied extensively under the more ambitious topic of program derivation, where one starts from a specification and derives hand-in-hand the program and the invariants justifying its correctness. Some references to this research are Dijkstra [1975, 1976], Gries [1981], Jones [1986], and Morgan [1990]. In our more modest endeavor of verification, where we assume that the program is already given, we still use some of the techniques

proposed by Dijkstra [1975] such as the *weakest precondition*.

**Refinement:** The topic of program refinement is another important subject which is not fully covered in this volume. In Section 1.4, we presented a special case of a refinement in which a multiple assignment statement in program MUX-PET1 is refined into two individual assignment statements, leading to program MUX-PET2. Section 1.5 presents an approach to transforming invariants that hold for the abstract version of the program (the program being refined) into invariants for the refined version. The general topic of refinement of sequential programs is discussed, among other places, in Back [1988], Jones [1986], Morris [1987], and Morgan [1990]. Refinement of reactive systems is considered in Lamport [1983a], Lynch and Tuttle [1987], Hoare, He, and Sanders [1987], Stark [1988], He [1989], Abadi and Lamport [1988], Jonsson [1990], Lynch [1990], Merritt [1990]. Additional methods proposed for verifying such refinements are presented in the collection "Stepwise Refinement of Distributed Systems," edited by de Bakker, de Roever, and Rozenberg [1990].

Surveys of verification techniques for concurrent and distributed programs, not restricted to techniques based on temporal logic, may be found in Barringer [1985] and Schneider and Andrews [1986].

**The examples:** Considering the main examples presented in this chapter, the *mutual-exclusion* problem was first formulated by Dijkstra [1965], together with a proposed solution which is proven to be *safe* (i.e., mutual exclusion is maintained) and deadlock free. As observed by Knuth [1966], this solution is not free of *individual starvation*. Knuth provided solutions that are free of individual starvation but allow one process to overtake its competitor an unbounded number of times. An improved solution was presented by de Bruijn [1967].

The first correct solution for two processes which overcomes all these problems is described by Dijkstra [1968a] and attributed to Dekker. The first fully correct solution for  $n$  processes, called the *bakery algorithm*, was presented by Lamport [1974]. It is safe, deadlock free, and starvation free with bounded overtaking, but is not finite state as it uses unbounded counters. An improved version was presented by Lamport [1976], where the question of *atomicity* of statements is examined more carefully. It is shown there that for  $n = 2$  the algorithm can be made finite state.

Algorithms MUX-PET1 and MUX-PET2 are based on the elegant solution of the mutual-exclusion problem for two processes presented in Peterson [1981]. The first finite-state solution for  $n$  processes, which ensures all the other requirements, was given by Peterson [1983] and is a generalization of Peterson [1981].

Many more algorithms for mutual exclusion are presented in Raynal [1986].

The *producer-consumer* problem was formulated by Dijkstra [1968a] and solved using semaphores.

## *Chapter 2*

# *Invariance: Applications*

Chapter 1 introduced the main method for proving that an assertion  $p$  is an invariant of program  $P$ . This method is essentially an induction on the positions in the computation. In this chapter we also identified the creative step of finding an inductive assertion that strengthens a given candidate invariant as the most difficult task in proofs of invariance. Some heuristics, such as supplementing assertions by preconditions, were proposed as techniques that may aid in the construction of inductive assertions.

This chapter concentrates on applications of the proof methods, considering more advanced examples of concurrent programs.

In Section 2.1, we consider parameterized programs, which consist of several similar processes whose number is determined by an input parameter.

In Sections 2.2 and 2.3, we consider some central paradigms of concurrent programs and verify their safety properties. Section 2.2 considers programs for allocation of a single resource among several competing processes. Section 2.3 considers the more general problem of multiple-resource allocation, in which each process may need several resources to perform its critical activity.

In Section 2.4, we present a method for algorithmic construction of invariants that involve linear expressions in the system variables. The method is guaranteed to find all linear invariants in the variables whose possible modifications are restricted to the addition (or subtraction) of a constant.

Section 2.5 shows that rule INV is complete for proving all state invariants.

Finally, Section 2.6 presents an algorithm for checking whether a given assertion is an invariant of a given finite-state program.

## 2.1 Parameterized Programs

Many distributed programs, in particular those that control communication and synchronization of networks, have as their body a parallel composition of many identical processes. Obviously, for any given number of such processes, the program is of the form we have been treating so far and is amenable to the analysis techniques discussed above.

For example, we may consider the simple case of many processes coordinating mutual exclusion by a single semaphore. In this case, the body  $S$  of each process may have the form

$$\ell_0: \text{loop forever do} \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } y \\ \ell_3: \text{critical} \\ \ell_4: \text{release } y \end{array} \right]$$

The complete program, for the special case of three processes, is given by

$$P^3 :: [\text{local } y: \text{integer where } y = 1; [S \parallel S \parallel S]]$$

Some renaming of locations may be needed to distinguish between the three copies of  $S$ .

However, establishing the correctness of this three-process program, i.e., showing that mutual exclusion is maintained, does not formally guarantee anything about the following similar program, which has four copies of the statement  $S$  as its processes:

$$P^4 :: [\text{local } y: \text{integer where } y = 1; [S \parallel S \parallel S \parallel S]]$$

In some cases, it is possible to treat the whole (infinite) family of programs  $P^1, P^2, P^3, P^4, \dots$  in a *uniform* way, providing a single proof that guarantees correctness of  $P^n$  for any  $n \geq 1$ . The key to such a uniform treatment is *parameterization*, i.e., presenting a single syntactic object that actually represents a family of objects. There are three areas that we have to parameterize: programs, their specifications, and the corresponding verification process.

### Parameterizing the Syntax of Programs

Programs may be parameterized by using compound statements of variable size. We mainly use variable-size versions of the compound constructs that are associative and commutative, i.e., the *cooperation* and *selection* statements. They appear in the following forms:

$$\prod_{j=1}^M S[j] \quad \text{and} \quad \bigvee_{j=1}^M S[j].$$

In both cases,  $S[j]$  is a statement, called a *parameterized* statement, in which  $j$  may appear explicitly as a variable in expressions and as a subscript in array references. In addition, the parameter  $j$  is considered to implicitly subscript all the variables that are declared local to  $S[j]$  and all the labels occurring within  $S[j]$ .

**Example** Consider program PAR-SUM of Fig. 2.1 that computes in  $z$  the sum of the squares of the elements of the array  $x[1..M]$ .

```

in  M: integer where M ≥ 1
      x : array [1..M] of integer
out z : integer where z = 0

      M
      || P[j] :: local y: integer
      j=1
                  [ l0: y := x[j]
                  [ l1: z := z + y · y
                  [ l2:

```

Fig. 2.1. Program PAR-SUM (parallel sum of squares).

The body of this program is a variable-size *cooperation* statement. The cooperation statement consists of a parallel composition of processes  $P[1], \dots, P[M]$ . As we see in this program, the parameter  $j$  appears explicitly in the reference to  $x[j]$ . It is also assumed to implicitly subscript all the labels appearing in  $P[j]$ , as well as the variable  $y$  which is locally declared in  $P[j]$ . In Fig. 2.2 we present the same program with the subscripting by  $j$  made explicit.

Usually, we prefer to present program PAR-SUM as in Fig. 2.1, but to reason about its behavior in terms of the fully subscripted version as in Fig. 2.2. ■

## Parameterized Transition Systems

Several extensions are necessary for the convenient representation of parameterized programs as transition systems. The main extension is the need to replace the notion of *finiteness* by that of *finite representability*.

For example, all transition systems we have considered so far have a fixed number of transitions. This is no longer true in the case of parameterized pro-

```

in  M: integer where M ≥ 1
      x : array [1..M] of integer
out z : integer where z = 0


$$\prod_{j=1}^M P[j] :: \begin{bmatrix} \text{local } y[j]: \text{integer} \\ \ell_0[j]: y[j] := x[j] \\ \ell_1[j]: z := z + y[j] \cdot y[j] \\ \ell_2[j]: \end{bmatrix}$$


```

Fig. 2.2. Program PAR-SUM-E — explicitly subscripted parameterized statements.

grams. Consider, for example, statement  $\ell_0$  in the program of Fig. 2.1. This statement copies the value of  $x[j]$  to the variable  $y$ , which is local to  $P[j]$ . Associating transitions with statements, we must have a separate transition  $\tau_0[j]$  for each  $j = 1, \dots, M$ . Since  $M$  is not fixed, there are unboundedly many transitions associated with the statement  $\ell_0$ . On the other hand, we can represent all these transitions by a single transition relation that uses  $j$  as a parameter. This transition relation is given by

$$\rho_{\ell_0}[j]: \text{move}(\ell_0[j], \ell_1[j]) \wedge y'[j] = x[j].$$

Consider a fully subscripted program, such as program PAR-SUM-E presented in Fig. 2.2; it is quite straightforward to define the transition system associated with it. Let us point out the main constituents of this transition system.

- *System Variables*

These consist of

$$\pi \quad M \quad x \quad z \quad y.$$

The variable  $\pi$  ranges over sets of locations, some of which may be subscripted, e.g., of the form  $\ell_1[j]$  for various  $j$ 's. Arrays, whether explicit (e.g.  $x$ ) or implicit (e.g.  $y$ ), are treated as variables that range over functions. Thus,  $x$  and  $y$  are considered as functions from the (subscript) domain  $[1..M]$  to the integers.

- *Transitions*

The transitions of the program consist of

$$\ell_0[j]: \text{place } x[j] \text{ in } y[j].$$

$$\ell_1[j]: \text{add } y[j] \cdot y[j] \text{ to } z.$$

- Initial Condition

$$\Theta: \pi = \{\ell_0[1], \dots, \ell_0[M]\} \wedge M \geq 1 \wedge z = 0.$$

We present below a computation of program PAR-SUM-E (and therefore also of program PAR-SUM) for the case where  $M = 2$  and  $x = \langle 1, 2 \rangle$ . For each state, we list the current values of  $\pi$ ,  $z$ , and  $y$ .

$$\begin{aligned} & \langle \{\ell_0[1], \ell_0[2]\}, 0, \langle -, - \rangle \rangle \xrightarrow{\ell_0[2]} \\ & \langle \{\ell_0[1], \ell_1[2]\}, 0, \langle -, 2 \rangle \rangle \xrightarrow{\ell_1[2]} \langle \{\ell_0[1], \ell_2[2]\}, 4, \langle -, 2 \rangle \rangle \xrightarrow{\ell_0[1]} \\ & \langle \{\ell_1[1], \ell_2[2]\}, 4, \langle 1, 2 \rangle \rangle \xrightarrow{\ell_1[1]} \langle \{\ell_2[1], \ell_2[2]\}, 5, \langle 1, 2 \rangle \rangle \end{aligned}$$

To retrieve the value of the  $k$ th element of any array  $y$ , we use the notation  $y[k]$  that can be viewed as applying the function  $y$  to the argument  $k$ . To represent the modification of an array  $x$ , we use the notation

$$update(x, k, e).$$

The value of this expression is an array that agrees with  $x$  on all subscript values  $i \neq k$ , and has as its  $k$ th element the value of  $e$ .

The main properties of the *update* function are

$$\begin{aligned} update(x, k, e)[k] &= e \\ j \neq k \rightarrow update(x, k, e)[j] &= x[j]. \end{aligned}$$

Note that even if the transition modifies  $k$ , the value to be used is that of  $k$  rather than  $k'$ .

Thus, the proper representation of the transition relation for  $\ell_0[j]$  is

$$\rho_0[j]: move(\ell_0[j], \ell_1[j]) \wedge y' = update(y, j, x[j]).$$

To simplify the notation, we write  $\rho_i[j]$  instead of  $\rho_{\ell_i}[j]$ , whenever it does not lead to confusion.

## Examples

We present two examples that illustrate the structure of parameterized programs. In later discussions, we will specify and verify these two examples.

### Example (multiple mutual exclusion by semaphores)

To illustrate the succinct way in which a parameterized program represents an infinite family of programs, consider the parameterized program MPX-SEM (Fig. 2.3) that achieves mutual exclusion by semaphores.

In this program we use incrementation modulo  $M$ , defined by

```

in      M: integer where M ≥ 2
local y : array [1..M] of integer
        where y[1] = 1, y[j] = 0 for 2 ≤ j ≤ M


$$\prod_{j=1}^M P[j] :: \begin{bmatrix} \ell_0: \text{loop forever do} \\ \ell_1: \text{noncritical} \\ \ell_2: \text{request } y[j] \\ \ell_3: \text{critical} \\ \ell_4: \text{release } y[j \oplus_M 1] \end{bmatrix}$$


```

Fig. 2.3. Program MPX-SEM (multiple mutual exclusion by semaphores).

$$j \oplus_M 1 = (j \bmod M) + 1 = \begin{cases} j + 1 & \text{if } j < M \\ 1 & \text{if } j = M \end{cases}$$

In Fig. 2.4 we present one member of the family, program MPX-SEM-2, which corresponds to  $M = 2$ . In this presentation, we explicitly represent the dependence of the locations on the processes' subscript. ■

### Example (finding the maximum of an array)

As another illustrative example, consider program MAX-ARRAY given in Fig. 2.5. This program places in the output variable  $z$  the maximum element of the integer array  $x$ . A special feature of this program is that the maximum value is computed in a highly parallel fashion.

The initial value of array  $y$  is specified to be  $y[j] = \top$  for all  $j = 1, \dots, M$ .

The first statement  $\ell_0$  spawns  $M^2$  processes. Each process  $P[i, j]$  sets the boolean variable  $y[i]$  to  $\text{F}$  if the array element  $x[i]$  is smaller than the element  $x[j]$ . After all the processes in the cooperation statement  $\ell_0$  have terminated, we are guaranteed that, for every  $i$  such that  $x[i]$  is not maximal in  $x$ ,  $y[i] = \text{F}$ . It follows that  $y[i] = \top$  iff  $x[i]$  is maximal.

The parallel processes spawned in  $\ell_4$  set  $z$  to  $x[i]$  if  $x[i]$  is maximal. Note that since there may be more than one  $i$  such that  $x[i]$  is maximal, more than one process may assign a value to  $z$ , but all the assigned values are the same.

The statement comprising the body of the MAX-ARRAY program may be inserted anywhere in a larger program, where we have an array  $x[1..M]$  and need to

**local**  $y$ : array [1..2] of integer **where**  $y[1] = 1$ ,  $y[2] = 0$

$$P[1] :: \left[ \begin{array}{l} \ell_0[1]: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1[1]: \text{noncritical} \\ \ell_2[1]: \text{request } y[1] \\ \ell_3[1]: \text{critical} \\ \ell_4[1]: \text{release } y[2] \end{array} \right] \end{array} \right]$$

||

$$P[2] :: \left[ \begin{array}{l} \ell_0[2]: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1[2]: \text{noncritical} \\ \ell_2[2]: \text{request } y[2] \\ \ell_3[2]: \text{critical} \\ \ell_4[2]: \text{release } y[1] \end{array} \right] \end{array} \right]$$

Fig. 2.4. Program MPX-SEM-2 — elaboration for  $M = 2$ .

**in**  $M$ : integer **where**  $M \geq 1$   
**x** : array [1.. $M$ ] of integer  
**local**  $y$  : array [1.. $M$ ] of boolean **where**  $y = T$   
**out**  $z$  : integer

$$\ell_0: \prod_{i,j \in [1..M]} P[i,j] :: \left[ \begin{array}{l} \ell_1: \text{if } x[i] < x[j] \text{ then } \ell_2: y[i] := F \\ \ell_3: \end{array} \right];$$

$$\ell_4: \prod_{i=1}^M Q[i] :: \left[ \begin{array}{l} \ell_5: \text{if } y[i] \text{ then } \ell_6: z := x[i] \\ \ell_7: \end{array} \right];$$

$$\ell_8:$$

Fig. 2.5. Program MAX-ARRAY (finding maximum of an array).

efficiently compute its maximum<sup>3</sup>. The variable  $M$  may assume different values in different entries to this statement. On each entry, the appropriate number of processes is spawned. ■

## Specifying Parameterized Programs

To specify and reason about parameterized programs, we introduce some conventions and special notation.

### Subscripted Locations

An important requirement for the specification of parameterized programs is the ability to refer to parameterized locations. We use integer variables to subscript locations, and allow the general integer operations and predicates to be applied to these variables.

Thus, to specify the property of mutual exclusion for program MPX-SEM (Fig. 2.3) for an arbitrary  $M$ ,  $M \geq 2$ , we may use the formula

$$i, j \in [1..M] \wedge i \neq j \rightarrow \square \neg(at\_\ell_3[i] \wedge at\_\ell_3[j]).$$

This formula states that for every two distinct processes  $P[i]$  and  $P[j]$ ,  $i, j \in [1..M]$ ,  $i \neq j$ , it is never the case that control is at  $\ell_3[i]$  and  $\ell_3[j]$  at the same time.

### Notations

Several notational conventions are useful for reasoning about parameterized programs. For a location  $\ell_i$  in a parameterized process we define the set  $L_i$  by

$$L_i = \{j \mid \ell_i[j] \in \pi\},$$

that is, the set of indices of the processes (a subset of  $\{1..M\}$ ) that currently reside at location  $\ell_i$ . We define a corresponding integer

$$N_i = |L_i|,$$

which represents the *number* of processes currently residing at location  $\ell_i$ .

For example, we may use this notation to specify mutual exclusion for program MPX-SEM of Fig. 2.3 by the formula

$$\square(N_3 \leq 1).$$

This formula states that the number of processes that may be concurrently executing at location  $\ell_3$  of their respective programs is at most 1.

---

<sup>3</sup> Program MAX-ARRAY is very efficient in terms of time, but is inefficient in terms of space requirements.

This notation is easily extended to refer to *location sets* rather than to individual locations. Thus,

$$\begin{aligned} L_{i_1, i_2, \dots, i_k} &= L_{i_1} \cup L_{i_2} \cup \dots \cup L_{i_k} \\ L_{i..j} &= L_i \cup L_{i+1} \cup \dots \cup L_j \\ N_{i_1, i_2, \dots, i_k} &= |L_{i_1, i_2, \dots, i_k}| \\ N_{i..j} &= |L_{i..j}|. \end{aligned}$$

## Verifying Parameterized Programs

Now that we have discussed the transition semantics of parameterized programs and introduced a sufficiently expressive assertion language, we can proceed with the verification of parameterized programs based on the principles advocated earlier, with very few extensions.

One consideration is the desire that checking the verification conditions over all transitions in the program should be done in a uniform way. Thus, when verifying that  $\{\varphi\} T \{\varphi\}$  holds for program MPX-SEM (Fig. 2.3), we have to check  $\{\varphi\} \tau \{\varphi\}$  for  $\tau = \ell_0[j], \ell_1[j], \ell_2[j], \ell_3[j], \ell_4[j]$  ( $1 \leq j \leq M$ ). For example, the verification condition for  $\tau = \ell_2[j]$  should use the transition relation

$$\rho_{\ell_2}[j]: \text{move}(\ell_2[j], \ell_3[j]) \wedge y[j] > 0 \wedge y' = \text{update}(y, j, y[j]-1).$$

Hopefully, we can establish in one proof that  $\{\varphi\} \ell_2[j] \{\varphi\}$  holds for all values of  $j$ .

We present several examples of uniform proofs establishing invariance properties of parameterized programs.

### Example (multiple mutual exclusion by semaphores)

The invariance property of program MPX-SEM (Fig. 2.3) is mutual exclusion, which can be specified by

$$\square \left( \underbrace{N_3 \leq 1}_{\psi} \right).$$

To prove this statement, we have to establish the invariance of the assertion  $\psi$ . To do so, we use the strengthening strategy and find an assertion which can be shown to be inductive. The assertion consists of the conjunction  $\varphi_1 \wedge \varphi_2$  of the following two assertions:

$$\varphi_1: \forall j: y[j] \geq 0$$

$$\varphi_2: N_{3,4} + \sum_{j=1}^M y[j] = 1.$$

Recall that  $N_{3,4}$  is the number of processes which currently execute at locations  $\ell_{3,4}$ .

There are two transitions that may affect  $\varphi_1$  or  $\varphi_2$ . They are  $\ell_2[i]$  and  $\ell_4[i]$ . We observe that  $\rho_2[i]$  implies

$$at\_\ell_2[i] \wedge at'\_ell_3[i] \wedge y[i] > 0 \wedge y'[i] = y[i] - 1 \wedge \forall j: j \neq i: y'[j] = y[j]$$

and  $\rho_4[i]$  implies

$$at\_\ell_4[i] \wedge at'\_ell_0[i] \wedge y'[i \oplus_M 1] = y[i \oplus_M 1] + 1 \wedge \forall j: j \neq i \oplus_M 1: y'[j] = y[j].$$

- $\varphi_1$  is inductive:

Clearly  $\Theta \rightarrow \varphi_1$ , that is

$$\underbrace{M \geq 2 \wedge \pi = \{\ell_0[1], \dots, \ell_0[M]\} \wedge y[1] = 1 \wedge y[2] = \dots = y[M] = 0}_{\Theta} \rightarrow \underbrace{\forall j: y[j] \geq 0}_{\varphi_1}.$$

It is easy to establish, for every  $i \in [1..M]$ ,

$$\ell_2[i]: \underbrace{\dots \wedge y[i] > 0 \wedge y'[i] = y[i] - 1 \wedge \forall j: j \neq i: y'[j] = y[j]}_{\rho_2[i]} \wedge \underbrace{\forall j: y[j] \geq 0}_{\varphi_1} \rightarrow \underbrace{\forall j: y'[j] \geq 0}_{\varphi'_1}.$$

$$\ell_4[i]: \underbrace{\dots \wedge y'[i \oplus_M 1] = y[i \oplus_M 1] + 1 \wedge \forall j: j \neq i \oplus_M 1: y'[j] = y[j]}_{\rho_4[i]} \wedge \underbrace{\forall j: y[j] \geq 0}_{\varphi_1} \rightarrow \underbrace{\forall j: y'[j] \geq 0}_{\varphi'_1}.$$

This establishes the preservation of  $\varphi_1$  over all transitions.

Consequently,  $\varphi_1$  is inductive and therefore invariant.

- $\varphi_2$  is inductive

Clearly  $\Theta \rightarrow \varphi_2$ , that is,

$$\underbrace{M \geq 2 \wedge \pi = \{\ell_0[1], \dots, \ell_0[M]\} \wedge y[1] = 1 \wedge y[2] = \dots = y[M] = 0}_{\Theta} \rightarrow \underbrace{N_{3,4} + \sum_{j=1}^M y[j] = 1}_{\varphi_2}$$

The implication holds since  $\pi = \{\ell_0[1], \dots, \ell_0[M]\}$  implies  $N_{3,4} = 0$  and the initial values  $y[1] = 1 \wedge y[2] = \dots = y[M] = 0$  imply  $\sum_{j=1}^M y[j] = 1$ .

The effects of  $\ell_2[i]$  and  $\ell_4[i]$  on  $\varphi_2$  can be expressed, respectively, by

$$\begin{aligned}\ell_2[i]: \quad N'_{3,4} &= N_{3,4} + 1 \quad \wedge \quad \sum_{j=1}^M y'[j] = \left( \sum_{j=1}^M y[j] \right) - 1 \\ \ell_4[i]: \quad N'_{3,4} &= N_{3,4} - 1 \quad \wedge \quad \sum_{j=1}^M y'[j] = \left( \sum_{j=1}^M y[j] \right) + 1.\end{aligned}$$

Consequently,  $\varphi_2$  is also inductive and therefore invariant.

Clearly,  $\varphi_1 \wedge \varphi_2$  implies

$$N_3 \leq N_{3,4} = 1 - \sum_{j=1}^M y[j] \leq 1,$$

showing that there can be at most one process at  $\ell_3$ .

This establishes mutual exclusion for program MPX-SEM for an arbitrary value of the parameter  $M$ ,  $M \geq 2$ . ■

### Example (finding the maximum of an array)

Consider program MAX-ARRAY of Fig. 2.5. The initial condition of the program is given by

$$\Theta: \quad \pi = \{\ell_0\} \wedge \bigwedge_{i=1}^M (y[i] = \text{t}).$$

We wish to prove the partial correctness of program MAX-ARRAY, which is given by the invariance of the assertion

$$\psi: \quad \text{at\_}\ell_8 \rightarrow \text{maximal}(z, x),$$

where  $\text{maximal}(z, x)$  states that  $z$  is the maximal element of the array  $x[1..M]$  and is an abbreviation for the formula:

$$\text{maximal}(z, x): \quad \exists k \in [1..M]: z = x[k] \wedge \forall j \in [1..M]: x[j] \leq z.$$

We base our proof on the property that, in any array, (at least) one of the elements is maximal. Let  $a \in [1..M]$  be the index of an element which is maximal in  $x$ . Thus,

$$\forall j \in [1..M]: x[j] \leq x[a].$$

We formulate several assertions that can be proved to be incrementally inductive, and hence invariant:

$$\begin{aligned}\varphi_1[i,j]: \quad & at\_{\ell_2}[i,j] \rightarrow x[i] < x[j] \\ \varphi_2[i]: \quad & at\_{\ell_6}[i] \rightarrow y[i] \\ \varphi_3: \quad & y[a] \\ \varphi_4[i]: \quad & x[i] \neq x[a] \rightarrow at\_{\ell_0} \vee at\_{\ell_{1,2}}[i,a] \vee \neg y[i] \\ \varphi_5: \quad & N_{0..4} > 0 \vee at\_{\ell_{5,6}}[a] \vee z = x[a]\end{aligned}$$

where  $N_{0..4}$  is the number of processes currently residing at locations  $\ell_0, \dots, \ell_4$ . The indices  $i$  and  $j$  range over  $1, \dots, M$ .

Assertion  $\varphi_1[i,j]$  claims that process  $P[i,j]$  can reach location  $\ell_2$  only if  $x[i] < x[j]$ .

Assertion  $\varphi_2[i]$  claims that process  $Q[i]$  can reach location  $\ell_6$  only if  $y[i]$  is true.

Assertion  $\varphi_3$  claims that  $y[a]$  remains true throughout the computation.

Assertion  $\varphi_4[i]$  claims that if  $x[i] \neq x[a]$  (and hence  $x[i] < x[a]$ ), then either the program is still at  $\ell_0$ , or process  $P[i,a]$  is still comparing  $x[i]$  with  $x[a]$  at  $\ell_1[i,a]$ , or is at the assignment  $\ell_2[i,a]$  that will eventually set  $y[i]$  to F, or  $y[i]$  has already been set to F. For the program to function correctly we must ensure that, by the time execution reaches  $\ell_4$ ,  $y[i]$  is false for all  $i$  such that  $x[i] \neq x[a]$  (and therefore  $x[i] < x[a]$ ). Assertion  $\varphi_4[i]$  guarantees this by stating that if  $x[i] \neq x[a]$  and  $y[i]$  is true then the execution of process  $P[i,a]$  has not terminated yet. Note that process  $P[i,a]$  is expected to set  $y[i]$  to false.

Assertion  $\varphi_5$  claims that either process  $Q[a]$ , which will eventually set  $z$  to  $x[a]$ , has not terminated yet, or  $z$  has already been set to  $x[a]$  (and will retain this value).

To conclude the partial correctness of the program, we observe that  $at\_{\ell_8}$  contradicts the two disjuncts  $N_{0..4} > 0$  and  $at\_{\ell_{5,6}}[a]$ , appearing in  $\varphi_5$ , leading to

$$at\_{\ell_8} \rightarrow z = x[a],$$

which, by the assumed maximality of  $x[a]$ , implies the invariance of

$at\_l_8 \rightarrow maximal(z, x).$

## 2.2 Single-Resource Allocation

An important problem in concurrent programming is that of *resource allocation*. A typical process in a concurrent system has some activity that it can perform with no need for coordination with other processes, while some other activity may require such coordination. Previously, we referred to these two kinds of activities as noncritical and critical, respectively.

Coordination is needed when there are some resources that must be used exclusively, i.e., by one process at a time, and these resources are shared between processes. For example, to print a long document on a printer requires uninterrupted possession of the printer by the printing task (process).

Here we deal with the simplest case of a single indivisible resource which is shared among  $M$  processes. When processes engage in their noncritical activity they do not need the resource. However, to start its critical activity, each process needs to acquire the resource. It is obvious that this simplest case of the resource-allocation problem is completely equivalent to that of mutual exclusion. As there is an indivisible resource, only one process at a time can engage in its critical activity.

### Solution 1: Semaphores

Allocation of a single resource among  $M$  processes or, equivalently, the  $M$ -process mutual-exclusion problem, was one of the earliest concurrent programming problems to be studied. Semaphores were specifically invented by E.W. Dijkstra as a synchronization device that solves the mutual-exclusion problem. Therefore, a simple solution to the single-resource allocation problem (as well as the  $M$ -processes mutual-exclusion problem) is that presented in Fig. 2.6. Note that program MUX-SEM of Fig. 1.6 (page 93) is an elaboration of RES-SEM for  $M = 2$ .

Using the notations for parameterized programs, the mutual-exclusion requirement for program RES-SEM can be stated as the invariance of the following assertion

$$\psi: N_3 \leq 1.$$

This invariant claims that at all times there can be at most one process  $P[j]$ ,  $j \in [1..M]$ , executing at location  $l_3$ .

Validity of  $\psi$  over program RES-SEM follows from the following invariant assertions which can be shown to be inductive:

**local**  $y$ : integer **where**  $y = 1$

$\prod_{j=1}^M P[j] ::$	$\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } y \\ \ell_3: \text{critical} \\ \ell_4: \text{release } y \end{array} \right] \end{array} \right]$
-------------------------	---

Fig. 2.6. Program RES-SEM (resource allocation by semaphores).

$$\varphi_0: y \geq 0$$

$$\varphi_1: N_{3,4} + y = 1.$$

Program RES-SEM in Fig. 2.6 provides a satisfactory solution to the resource-allocation problem for systems (architectures) in which the semaphore construct is available. We proceed to study alternative solutions for architectures that do not provide semaphores.

## Solution 2: Synchronous Message Passing

The first case we consider is an architecture in which processes communicate by synchronous message passing. In Fig. 2.7 we present a proposed solution for this case. Execution of this program starts with all processes positioned at their initial locations. Thus, the initial value of  $\pi$  is  $\{m_0, \ell_0[1], \dots, \ell_0[M]\}$ .

Program RES-ND appoints a special process  $A$  to be the resource allocator, sometimes referred to as the arbiter. Nondeterministic process  $A$  centrally manages the allocation of the resource among  $M$  customer processes  $C[1], \dots, C[M]$  which compete for the resource.

The allocator and customers communicate via an array of synchronous channels  $\alpha[1..M]$ . The allocator has an  $M$ -wide selection statement at  $m_1$ , open to communicate with all the customers on their corresponding channels. On exiting the noncritical section, customer  $C[i]$  attempts to send the value 1 on channel  $\alpha[i]$ . The message is sent only with the cooperation of the allocator at  $m_1$ , which has to select communication with that particular  $C[i]$ . When the message is transmitted, the customer proceeds to its critical section, while the allocator moves to  $m_2$ , awaiting the next message on the same channel. On exiting the critical section,  $C[i]$  sends the message 0 on  $\alpha[i]$ . This releases the allocator which returns to  $m_1$  to await communication with the next customer.

```

in      M: integer where M > 0
local α : array [1..M] of integer channel

A ::   
$$\left[ \begin{array}{l} \text{local } d: \text{integer where } d = 0 \\ m_0: \text{loop forever do} \\ \left[ \begin{array}{l} m_1: \bigvee_{j=1}^M [\alpha[j] \Rightarrow d; m_2: \alpha[j] \Rightarrow d] \end{array} \right] \end{array} \right]$$

||


$$\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \alpha[i] \Leftarrow 1 \\ \ell_3: \text{critical} \\ \ell_4: \alpha[i] \Leftarrow 0 \end{array} \right] \end{array} \right]$$


$$\prod_{i=1}^M C[i] ::$$


```

Fig. 2.7. Program RES-ND (resource allocator) — nondeterministic message-passing version.

There is nothing that prevents the allocator from granting the resource to the same  $C[i]$  twice in succession. However, in the long run, the requirement of compassion associated with the communication statements prevents the allocator from ignoring a given customer forever. Thus, every customer waiting patiently at location  $\ell_2$  is guaranteed to eventually be admitted to its critical section.

The main safety property of program RES-ND is that of mutual exclusion, which is stated by the invariance of the following assertion

$$\psi: \bigwedge_{k \neq n} \neg(at\_\ell_3[k] \wedge at\_\ell_3[n]).$$

The invariance of  $\psi$  follows from the invariance of the following assertion

$$\varphi[i]: at\_\ell_{3,4}[i] \leftrightarrow at\_m_2[i],$$

stated for each  $i = 1, \dots, M$ . This assertion states that customer  $C[i]$  can be in its (extended) critical range  $[\ell_3, \ell_4]$  if and only if, at the same time, the allocator is at location  $m_2$ , having selected branch  $j = i$  of the multiple selection statement  $m_1$ . It is straightforward to check that  $\varphi[i]$  is inductive for each  $i \in [1..M]$ .

Consider  $k, n \in [1..M]$ , where  $k \neq n$ . By invariants  $\varphi[k]$  and  $\varphi[n]$ ,  $at\_\ell_3[k] \wedge at\_\ell_3[n]$  implies  $at\_m_2[k] \wedge at\_m_2[n]$ . Since locations  $m_2[k]$  and  $m_2[n]$  are con-

flicting for  $k \neq n$ , this shows that  $\text{at\_}\ell_3[k] \wedge \text{at\_}\ell_3[n]$  is contradictory, establishing  $\psi$ .

Like RES-SEM, the accessibility property that program RES-ND can guarantee to each of its customers is very weak. It only promises that every waiting process will eventually be served. In some cases, we need a more binding commitment. For example, it would be nice if a solution to the resource-allocation problem could guarantee that, from the time a customer  $C[i]$  starts waiting at  $\ell_2$ , each of the other  $M - 1$  processes can enter their critical sections ahead of  $C[i]$  at most once. While our model cannot measure real time, a guarantee such as the one described would imply a worst-case waiting time that is proportional to  $M$ .

The next proposed solution (for the synchronous message-passing architecture) aims to achieve this sharper accessibility guarantee.

### **Solution 3: Bounded Overtaking**

Program RES-MP, presented in Fig. 2.8, implements a stricter customer selection discipline than the simple nondeterministic selection of program RES-ND.

Like program RES-ND, program RES-MP consists of an allocator process  $A$  and customer processes  $C[1], \dots, C[M]$ . As before, allocator and customer  $C[i]$  communicate via the synchronous channel  $\alpha[i]$  to negotiate the ownership of the resource.

New to program RES-MP is the explicit round-robin order in which the allocator scans for customer requests. This is implemented by variable  $t$  which is incremented in cyclic order. At any point,  $t \in [1..M]$  indicates the index of the process that the allocator is currently considering. To test whether customer  $C[t]$  has posted a request (waiting at  $\ell_2$ ), the program uses a *prefer-to* statement of the form

**prefer  $S_1$  to  $S_2$ .**

In the case that only one of  $S_1$  and  $S_2$  is enabled, the *prefer-to* statement allows (similar to the *selection* statement) the enabled statement to start its execution while discarding its competitor. If both  $S_1$  and  $S_2$  are enabled, then  $S_1$  is preferred to  $S_2$  and only  $S_1$  can start executing.

In the program for  $A$ , execution of  $m_1^a$  (discarding  $m_1^b$ ) can start whenever  $m_1^a$  is enabled. Statement  $m_1^b$  can be executed (discarding  $m_1^a, m_2$ ) only if  $m_1^a$  is presently disabled, i.e.,  $C[t]$  is not currently waiting at  $\ell_2$ .

According to this strict algorithm, the allocator scans the processes in cyclic order. Statements  $m_2$  and  $m_1^b$  use incrementation modulo  $M$  in order to move from one process to the next in cyclic order. Whenever, on its cyclic scan, the allocator detects a requesting customer, it grants the resource to the customer and waits for a release. When the resource is released, the allocator resumes scanning.

```

in   M: integer where  $M > 0$ 
local  $\alpha$  : array [1..M] of integer channel

A :: 
$$\left[ \begin{array}{l} \text{local } d, t: \text{integer where } t = 1 \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \text{prefer} \\ \left[ \begin{array}{l} m_1^a: \alpha[t] \Rightarrow d \\ m_2: (\alpha[t] \Rightarrow d; t := t \oplus_M 1) \end{array} \right] \\ \text{to} \\ m_1^b: t := t \oplus_M 1 \end{array} \right] \\ || \\ \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \alpha[i] \Leftarrow 1 \\ \ell_3: \text{critical} \\ \ell_4: \alpha[i] \Leftarrow 0 \end{array} \right] \end{array} \right] \\ \prod_{i=1}^M C[i] :: \end{array} \right]$$


```

Fig. 2.8. Program RES-MP (resource allocator) — message-passing version.

### Proving Mutual Exclusion

The property of mutual exclusion for program RES-MP is stated by the invariance of the assertion

$$\psi: \neg(at\_\ell_3[k] \wedge at\_\ell_3[n]) \quad \text{for every } k, n \in [1..M], k \neq n,$$

which claims that at most one process may be executing at  $\ell_3$  at any given time.

Intuitively, mutual exclusion is based on the following two obvious facts:

- (a) A customer may enter its critical section only when permitted by the allocator.
- (b) The allocator permits entry to only one customer at a time.

These facts may be formally expressed by the assertion

$$\varphi[i]: at\_\ell_{3,4}[i] \leftrightarrow at\_{m_2} \wedge t = i,$$

stated for each  $i \in [1..M]$ . Clearly, the validity of this assertion implies mutual

exclusion. This is because if both  $at\_l_3[k]$  and  $at\_l_3[n]$  hold for  $k \neq n$  then, by  $\varphi[k]$  and  $\varphi[n]$ ,  $t$  is equal to both  $k$  and  $n$  at the same time, which is impossible.

Let us consider an arbitrary  $i$  and prove that  $\varphi[i]$  is inductive. Initially,  $at\_l_{3,4}[i] = at\_m_2 = F$  so both sides of  $\varphi[i]$  are false.

The transitions that appear to threaten the validity of  $\varphi[i]$  are  $l_2[i]$ ,  $l_4[i]$ ,  $m_1^a$ , and  $m_2$ . Clearly,  $l_2[i]$  and  $l_4[i]$  can be taken only together with matching communication transitions. The possible matching transitions are  $m_1^a$  and  $m_2$  while  $t = i$ . Consequently, we consider:

- $\langle l_2[i], m_1^a \rangle$  while  $t = i$ : sets both  $at\_l_{3,4}[i]$  and  $at\_m_2 \wedge t = i$  to T.
- $\langle l_4[i], m_2 \rangle$  while  $t = i$ : sets both  $at\_l_{3,4}[i]$  and  $at\_m_2 \wedge t = i$  to F.

It follows that  $\varphi[i]$  is an invariant of the program for every  $i \in [1..M]$ .

This concludes the proof that  $\psi$  is an invariant of program RES-MP, ensuring that the program maintains mutual exclusion.

We may consider what changes are necessary in order to move from synchronous message passing to asynchronous message passing. In **Problem 2.1**, the reader is requested to propose asynchronous versions for these two programs and prove that the proposed programs maintain mutual exclusion.

## Solution 4: Shared Variables

Next, we consider an architecture in which processes communicate by shared variables but the semaphore construct is not available. This is the architecture in which programs such as MUX-PET1 (Fig. 1.13, page 121) and MUX-PET2 (Fig. 1.14, page 124) are relevant and provide good solutions for mutual exclusion between two processes.

Program RES-SV, presented in Fig. 2.9, is proposed as a solution to the resource-allocation problem in a shared-variables architecture.

In this program the allocator and customers communicate via two boolean arrays,  $r[1..M]$  and  $g[1..M]$ , both initialized to F. Customer  $C[i]$  sets  $r[i]$  to T to signal a *request* for the resource, and resets  $r[i]$  to F to signal a *release* of the resource. The allocator sets  $g[i]$  to T to notify customer  $C[i]$  that the requested resource has been *granted* to it. The allocator resets  $g[i]$  to F to *acknowledge* to  $C[i]$  its release of the resource.

One cannot fail to detect the strong similarity between program RES-SV and program RES-MP. In both programs the allocator scans the customers in a cyclic order, granting the resource to the first requesting customer it encounters. The main differences between the two programs are due to the different means of communication they have at their disposal. In program RES-MP, the allocator becomes aware that  $C[i]$  is awaiting service and grants  $C[i]$  the resource, all in

in  $M$  : integer where  $M > 0$   
 local  $g, r$ : array [1.. $M$ ] of boolean where  $g = F, r = F$

$$A :: \left[ \begin{array}{l} \text{local } t: \text{integer where } t = 1 \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{if } r[t] \text{ then} \\ \quad \left[ \begin{array}{l} m_2: g[t] := T \\ m_3: \text{await } \neg r[t] \\ m_4: g[t] := F \end{array} \right] \\ m_5: t := t \oplus_M 1 \end{array} \right] \end{array} \right]$$

||

$$\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: r[i] := T \\ \ell_3: \text{await } g[i] \\ \ell_4: \text{critical} \\ \ell_5: r[i] := F \\ \ell_6: \text{await } \neg g[i] \end{array} \right] \end{array} \right]$$

$$\left[ \begin{array}{l} M \\ || \\ i=1 \end{array} \right] C[i] ::$$

Fig. 2.9. Program RES-SV (resource allocator) — shared-variables version.

the single joint transition  $\langle m_1^a, \ell_2 \rangle$ . Program RES-SV needs at least four transitions to achieve the same effect:

1.  $C[i]$  sets  $r[i]$  to  $T$  and moves to  $\ell_3$ .
2.  $A$  recognizes in  $m_1$  that  $r[i] = T$  and moves to  $m_2$ .
3.  $A$  sets  $g[i]$  to  $T$  and moves to  $m_3$ .
4.  $C[i]$  finds out in  $\ell_3$  that  $g[i] = T$  and moves to  $\ell_4$ .

Mutual exclusion for program RES-SV is stated by the invariance of the assertion

$$\psi: \neg(at\_\ell_4[k] \wedge at\_\ell_4[n]) \quad \text{for } k, n \in [1..M], k \neq n.$$

We will establish a sequence of inductive assertions that will lead up to the invariance of  $\psi$ .

In the analysis of program RES-MP we only needed a single auxiliary invariant, which we rename here  $\chi[i]$ :

$$\chi[i]: \quad at_{-\ell_3,4}[i] \leftrightarrow at_{-m_2} \wedge t = i.$$

This invariant was helpful since it identified a critical region within the allocator, namely location  $m_2$ , and a critical region within customer process  $C[i]$ , namely  $[\ell_3, \ell_4]$ , and established a strong correspondence between the two. Invariant  $\chi[i]$  states that  $C[i]$  is critical (i.e. executes in the critical section  $\ell_{3,4}$ ) iff  $A$  is critical while paying attention to  $C[i]$  ( $t = i$ ).

No such simple correspondence can be established for program RES-SV. The critical region within  $A$  is obviously  $m_3$ . Suppose  $A$  is at  $m_3$  with  $t = i$ ; what can we say about the current location of  $C[i]$ ? It could be anywhere within  $\{\ell_3, \ell_4, \ell_5, \ell_6\}$ . Clearly,  $\ell_6$  should not be considered critical since the program allows a situation in which  $C[i]$  is at  $\ell_6$  while  $C[i \oplus_M 1]$  is already at  $\ell_4$ .

Instead of establishing a direct correspondence between the locations of  $A$  and those of  $C[i]$ , we establish connections between the locations of the allocator and the customers and the values of  $r[i]$ ,  $g[i]$ .

## Two Equivalences

The invariant

$$\varphi_1[i]: \quad at_{-m_3,4} \wedge t = i \leftrightarrow g[i]$$

can be established for every  $i \in [1..M]$ . The transitions that may affect the variables mentioned in this assertion are  $m_2$ ,  $m_4$ , and  $m_5$ . Transitions  $m_2$  and  $m_4$  set  $at_{-m_3,4}$  and  $g[t]$  to true and false, respectively. Since under  $\rho_{m_5}$  both  $at_{-m_3,4}$  and  $at'_{-m_3,4}$  are false, the boolean value of  $at_{-m_3,4} \wedge t = i$  is not changed by  $m_5$ .

The invariant

$$\varphi_2[i]: \quad at_{-\ell_3,5}[i] \leftrightarrow r[i]$$

can also be established for every  $i \in [1..M]$ . The only relevant transitions are  $\ell_2[i]$  and  $\ell_5[i]$ . Transition  $\ell_2[i]$  sets both  $at_{-\ell_3,5}[i]$  and  $r[i]$  to true, while transition  $\ell_5[i]$  resets both  $at_{-\ell_3,5}[i]$  and  $r[i]$  to false.

## Four Implications

As seen in invariants  $\varphi_1$  and  $\varphi_2$ , the control location of  $A$  and the value of  $t$  uniquely determine the value of  $g[i]$ , and the control location of  $C[i]$  determines the value of  $r[i]$ . In contrast, the values of  $r[i]$  and  $g[i]$  are not fully determined by the control locations of  $A$  and  $C[i]$ , respectively. Here we have only one-way implications, given by

$$\varphi_3: \quad at_{-m_2} \rightarrow r[t]$$

$$\varphi_4: \quad at_{-m_4} \rightarrow \neg r[t]$$

- $\varphi_5[i]: \text{at\_}\ell_{0..2}[i] \rightarrow \neg g[i]$
- $\varphi_6[i]: \text{at\_}\ell_{4,5}[i] \rightarrow g[i].$

We denote by  $\varphi_5$  and  $\varphi_6$  the conjunctions  $\bigwedge_{i=1}^M \varphi_5[i]$  and  $\bigwedge_{i=1}^M \varphi_6[i]$ , respectively.

Initially, assertions  $\varphi_3$ ,  $\varphi_4$ , and  $\varphi_6$  hold trivially since their antecedent is false. Assertion  $\varphi_5$  holds initially since  $\text{at\_}\ell_0[i]$  is true and  $g[i]$  is false.

We will show that the conjunction  $\varphi_3 \wedge \varphi_4 \wedge \varphi_5 \wedge \varphi_6$  is preserved under all transitions. It is interesting to note that while none of  $\varphi_3, \dots, \varphi_6$  is inductive on its own, their conjunction is inductive. For each of the conjuncts, we consider the transitions that may affect its validity.

#### *Preservation of $\varphi_3$*

The transitions relevant to  $\varphi_3$  are

- $m_1^T$  — can be taken only if  $r[t]$  holds.
- $\ell_5[t]$  while  $\text{at\_}m_2$ . By  $\varphi_6$ ,  $\ell_5[t]$  is enabled only if  $g[t]$  is currently true, which contradicts  $\text{at\_}m_2$  according to  $\varphi_1[t]$ .

#### *Preservation of $\varphi_4$*

The relevant transitions are

- $m_3$  — possible only if  $r[t]$  is false.
- $\ell_2[t]$  while  $\text{at\_}m_4$ . By  $\varphi_5[t]$ ,  $\ell_2[t]$  can be taken only if  $g[t]$  is false. This contradicts  $\text{at\_}m_4$  according to  $\varphi_1[t]$ .

#### *Preservation of $\varphi_5$*

The relevant transitions are

- $\ell_6[i]$  — possible only if  $g[i]$  is false.
- $m_2$  while  $i = t$  and  $\text{at\_}\ell_{0..2}[i]$ . By  $\varphi_3$ , this is possible only if  $r[t]$  is true which, in view of  $\varphi_2[t]$ , contradicts  $\text{at\_}\ell_{0..2}[i]$ .

#### *Preservation of $\varphi_6$*

The relevant transitions are

- $\ell_3[i]$  — possible only if  $g[i]$  is true.
- $m_4$  while  $i = t$  and  $\text{at\_}\ell_{4,5}[i]$ . By  $\varphi_4$ , this is possible only if  $r[t]$  is false which, in view of  $\varphi_2[t]$ , contradicts  $\text{at\_}\ell_{4,5}[i]$ .

Invariants  $\varphi_1\text{-}\varphi_6$  can be summarized in the diagram of Fig. 2.10.

The diagram can be viewed as representing the disjunctive assertion:

$$\varphi[i]: \text{idle}[i] \vee \text{requesting}[i] \vee \text{granted}[i] \vee \text{releasing}[i],$$

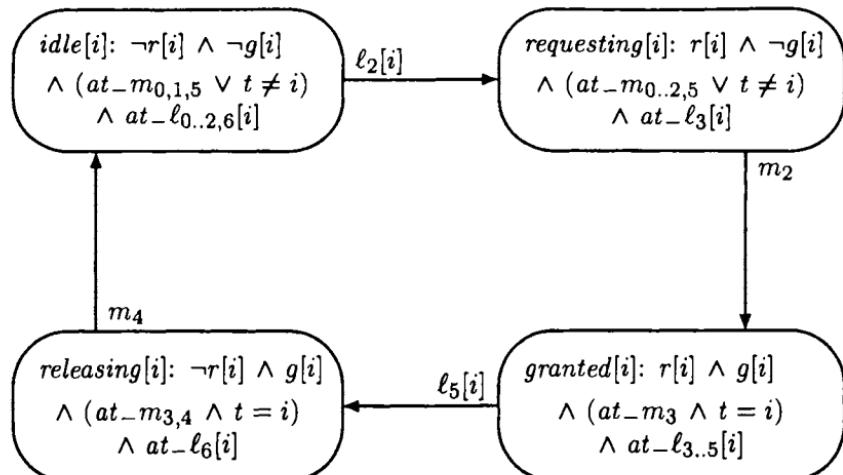


Fig. 2.10. Four phases in the life of a customer process.

which states that the execution of process  $C[i]$  must be in one of four identified phases. These four phases form a protocol through which process  $C[i]$  cycles as shown in the diagram. Directed edges in the diagram identify the transitions that transform a state satisfying one of the disjuncts into a state satisfying the next disjunct.

It is not difficult to see that the invariance of assertions  $\varphi_1 - \varphi_6$  is equivalent to the invariance of the single assertion  $\varphi$ .

### Mutual Exclusion

Mutual exclusion can be established from  $\varphi_6$  and  $\varphi_1$  or, alternately, directly from  $\varphi$ . Assume that  $\text{at\_}\ell_4[k]$  and  $\text{at\_}\ell_4[n]$  hold for  $k \neq n$ . By  $\varphi_6[k]$  and  $\varphi_6[n]$ , it follows that  $g[k] = g[n] = \tau$ . By  $\varphi_1[k]$  and  $\varphi_1[n]$ , this implies that  $t = k$  and  $t = n$  which is impossible. ■

In **Problem 2.2**, the reader is requested to verify a parameterized program that implements mutual exclusion, based on a central manager process which arbitrates between competing customer processes. In **Problem 2.3**, the reader is requested to consider a generalized version of Peterson's algorithm, implementing mutual exclusion among  $N$  processes.

### The Readers-Writers Problem

As another example of allocation of a single resource, consider the readers-writers

problem. This problem generalizes the mutual-exclusion and the single-resource allocation problems by considering two types of critical sections, a *reading section* and a *writing section*.

This distinction is motivated by the situation in which we wish to coordinate an access to a shared variable or data structure between several processes. We may allow several processes to simultaneously read parts of the structure. But when the structure is being modified by one of the processes, we want to exclude all other processes from accessing the structure. Obviously, we want to exclude other writers because their actions may interfere with the actions of the first writer and lead to an ill-defined structure. We also want to exclude other readers, because they may obtain an inconsistent view of the structure, reading some parts before, and other parts after, their modification by a writer process.

A simple solution to the readers-writers problem may be obtained by a generalization of the semaphore statements. We introduce semaphore statements with two arguments

**request** ( $y, c$ )      and      **release** ( $y, c$ ),

where  $c$  is a positive integer constant. The transition semantics of these statements are equivalent to those of the grouped statements

$\langle \text{await } y \geq c; y := y - c \rangle$       and       $\langle y := y + c \rangle$ ,

respectively.

Program READ-WRITE of Fig. 2.11 presents a simple solution to the readers-writers problem that uses generalized semaphores.

Program READ-WRITE consists of  $M$  identical processes. Each process is a single loop that alternates between a noncritical section and a nondeterministic choice between performing a read-protocol  $R$  and a write-protocol  $W$ . The nondeterministic choice in program READ-WRITE schematically represents the fact that, on exit from  $\ell_1$ , the process sometimes proceeds to  $\ell_2$ , and at other times it proceeds to  $\ell_5$ .

The readers-writers problem can be specified by:

$$\forall i, j \in [1..M]: i \neq j: \quad \square \underbrace{at\_l_6[i] \rightarrow \neg(at\_l_6[j] \vee at\_l_3[j])}_{\psi[i,j]} .$$

While this formula contains explicit quantification and is therefore not a state-quantified formula, its  $P$ -validity can be established by proving the  $P$ -validity of  $\square \psi[i, j]$  for every  $i, j \in [1..M], i \neq j$ .

Since the readers-writers problem is a generalization of the mutual-exclusion problem and the solution uses a generalization of semaphores, it is not very surprising that the assertions we suggest as a strengthening of  $\psi[i, j]$  are generalizations of the assertions we used in the mutual-exclusion case:

in  $M$ : integer where  $M \geq 1$   
 local  $y$  : integer where  $y = M$

$$\prod_{i=1}^M P[i] :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \quad \left[ \begin{array}{l} R :: \left[ \begin{array}{l} \ell_2: \text{request } (y, 1) \\ \ell_3: \text{read} \\ \ell_4: \text{release } (y, 1) \end{array} \right] \\ \text{or} \\ W :: \left[ \begin{array}{l} \ell_5: \text{request } (y, M) \\ \ell_6: \text{write} \\ \ell_7: \text{release } (y, M) \end{array} \right] \end{array} \right] \end{array} \right] \end{array} \right]$$

Fig. 2.11. Program READ-WRITE (readers-writers with generalized semaphores).

$$\varphi_1: y \geq 0$$

$$\varphi_2: N_{3,4} + M \cdot N_{6,7} + y = M,$$

where  $N_{3,4}$  stands for the number of processes currently executing at  $\ell_3$  or  $\ell_4$ , and similarly for  $N_{6,7}$ .

- $\varphi_1$  is invariant:

It is easy to verify that  $\varphi_1$  is inductive and therefore invariant.

- $\varphi_2$  is invariant:

To check that  $\varphi_2$  is inductive, we observe that initially  $N_{3,4} = N_{6,7} = 0$ , and  $y = M$ , which imply  $\varphi_2$ .

There are four transitions that modify the expressions on which  $\varphi_2$  depends:

- Transition  $\ell_2[i]$  preserves  $N_{3,4} + y$ , as  $N'_{3,4} = N_{3,4} + 1$  and  $y' = y - 1$ .
- Transition  $\ell_4[i]$  preserves  $N_{3,4} + y$ , as  $N'_{3,4} = N_{3,4} - 1$  and  $y' = y + 1$ .
- Transition  $\ell_5[i]$  preserves  $M \cdot N_{6,7} + y$ , as  $N'_{6,7} = N_{6,7} + 1$  and  $y' = y - M$ .
- Transition  $\ell_7[i]$  preserves  $M \cdot N_{6,7} + y$ , as  $N'_{6,7} = N_{6,7} - 1$  and  $y' = y + M$ .

Thus,  $\varphi_1 \wedge \varphi_2$  is invariant. It is easy to show that  $\varphi_1 \wedge \varphi_2$  implies

$$N_{6,7} > 0 \rightarrow (N_{6,7} = 1 \wedge N_{3,4} = 0),$$

from which  $\psi[i, j]$  immediately follows.

In **Problem 2.4**, we ask the reader to show that grouped statements containing communication statements can sometimes be ungrouped. In **Problem 2.5**, the reader is requested to verify the correctness of a program for resource allocation between processes arranged in a ring.

## 2.3 Multiple-Resource Allocation

We have considered previously the problem of allocation of a single resource between several competing processes. The readers-writers problem can be viewed as a generalization of the above problem, still competing for a single resource, but having different modes of utilization of the resource, some of which are not exclusive.

A more general situation is one in which we have several processes and resources, and each process needs several of the resources to perform its critical activity.

### Three Resources for Three Processes

Consider the special case of three processes  $P_1$ ,  $P_2$ ,  $P_3$ , and three resources,  $R_1$ ,  $R_2$ ,  $R_3$ . The requirements of each process are summarized in the table below:

Needs Process	$R_1$	$R_2$	$R_3$
$P_1$		+	+
$P_2$	+		+
$P_3$	+	+	

A possible solution to this synchronization problem, which also indicates the general pattern of such solutions, is presented by program RES-3 of Fig. 2.12, which deals with 3 resources. This program uses three semaphore variables,  $r_1$ ,  $r_2$ ,  $r_3$ , to represent the availability of the resources  $R_1$ ,  $R_2$ ,  $R_3$ . Initially all resources are available, represented by  $r_1 = r_2 = r_3 = 1$ . Then, as each process decides to enter its critical section, it requests the resources it needs, one at a time, by issuing a *request* statement addressed to the corresponding semaphore variable. Each successful *request*  $r_j$  statement performed by process  $P_i$  appropriates the resource  $R_j$  for  $P_i$ . Until the resource is released, no other process can obtain it. Following a successful acquisition of the resources it needs, the process enters its critical section. On exit from the critical section, the process releases all the resources it previously acquired.

```

local r1, r2, r3: integer where r1 = r2 = r3 = 1

P1 :: [l0: loop forever do
          [l1: noncritical
           l2: request r2; l3: request r3
           l4: critical
           l5: release r2; l6: release r3 ] ]
          ||

P2 :: [m0: loop forever do
          [m1: noncritical
           m2: request r1; m3: request r3
           m4: critical
           m5: release r1; m6: release r3 ] ]
          ||

P3 :: [k0: loop forever do
          [k1: noncritical
           k2: request r1; k3: request r2
           k4: critical
           k5: release r1; k6: release r2 ] ]

```

Fig. 2.12. Program RES-3 — three processes needing three resources.

The general exclusion property of a multiple-resource situation is that no two processes that need the same resource may simultaneously reside in their critical sections. In the specific example of program RES-3, this forces exclusion between every two of the three processes, since every two have precisely one needed resource in common.

As usual, exclusion is guaranteed by invariants, one for each resource. It is not difficult to check that the following assertions are inductive:

$$\begin{aligned}
 \varphi_0: \quad & r_1 \geq 0 \wedge r_2 \geq 0 \wedge r_3 \geq 0 \\
 \varphi_1: \quad & r_1 + at\_m_{3..5} + at\_k_{3..5} = 1 \\
 \varphi_2: \quad & r_2 + at\_{l_{3..5}} + at\_{k_{4..6}} = 1
 \end{aligned}$$

$$\varphi_3: r_3 + at\_l_{4..6} + at\_m_{4..6} = 1.$$

It is easy to see that these invariants ensure pairwise exclusion between the processes. For example, to show exclusion between  $P_1$  and  $P_3$ , we examine the invariant  $\varphi_2$ , corresponding to the resource  $R_2$  common to  $P_1$  and  $P_3$ . From  $\varphi_2$  we obtain

$$at\_l_4 + at\_k_4 \leq at\_l_{3..5} + at\_k_{4..6} \leq 1,$$

which shows that  $P_1$  and  $P_3$  cannot be at their critical sections at the same time.

### The Order of Requesting Resources

The example above points to a general solution to the multiple-resource problem using semaphores. For each process  $P_j$  in the system, let  $R_{i_1}, R_{i_2}, \dots, R_{i_k}$  be the list of resources  $P_j$  needs. Then the program for  $P_j$  is the one presented in Fig. 2.13.

```

local  $r_1, r_2, \dots, r_n$ : integer where  $r_1 = r_2 = \dots = r_n = 1$ 

||  $P_j ::$  [loop forever do
    [noncritical
     request  $r_{i_1}$ 
     :
     request  $r_{i_k}$ 
     critical
     release  $r_{i_1}$ 
     :
     release  $r_{i_k}$ ]
    ] ||
```

Fig. 2.13. Program for a general process needing resources.

We can generalize our proof of exclusion to this general program.

An important requirement for the correctness of this algorithm is that the *order* in which each process requests the resources be consistent with some global order. This is easily achievable by ordering the resources in some fixed order  $R_1, \dots, R_n$ , and then requiring the processes to request the resources in ascending order, i.e.,  $i_1 < i_2 < \dots < i_k$ , as is done in program RES-3 (Fig. 2.12).

To see how essential this restriction is, consider a version of the program of Fig. 2.12 in which we merely interchange the statements  $m_2$  and  $m_3$ , so that the requesting segment of  $P_2$  now reads

$$\tilde{m}_2: \text{request } r_3; \quad \tilde{m}_3: \text{request } r_1.$$

This seemingly innocuous change introduces the possibility of deadlock into the system. Consider a computation in which  $P_1$  has progressed up to  $\ell_3$ , acquiring  $R_2$ ,  $P_2$  has progressed up to  $\tilde{m}_3$ , acquiring  $R_3$ , and  $P_3$  has progressed up to  $k_3$ , acquiring  $R_1$ . At this point, all semaphore variables are zero, and each process waits in front of another *request* statement, unable to move. This is deadlock.

On the other hand, we argue that if all processes request their resources in ascending order, no deadlock can occur.

Assume that all processes request their needed resources in ascending order, and a particular state is claimed to be a deadlock situation. This can happen only if all processes are in front of a *request* statement. We can assume that some processes have already acquired some of the resources. Let  $R_j$  be the acquired resource with the highest index, and let  $P_k$  be the process that acquired  $R_j$ . Process  $P_k$  must be waiting in front of a **request**  $r_i$  for some  $i > j$ , since resources are requested in ascending order. Why cannot  $P_k$  proceed? Only because some other process has already acquired  $R_i$ . But this is impossible since we assumed that  $j$ ,  $j < i$ , was the highest index of any acquired resource. This shows that if all resources are requested in ascending order, deadlock cannot occur. Note that the order in which resources are released is immaterial.

## The Dining Philosophers Problem

A famous special case of multiple-resource allocation is known as the dining philosophers problem. As originally described by Dijkstra,  $M$  philosophers are seated at a round table. Each philosopher alternates between a thinking phase and a phase in which he becomes hungry and wishes to eat. There are  $M$  chopsticks placed around the table, one chopstick between every two philosophers, as depicted in Fig. 2.14.

Since their staple food is rice, each philosopher needs to acquire the chopsticks on his left and on his right in order to eat. A chopstick can be possessed by only one philosopher at a time.

The dining philosophers problem was designed to illustrate a multiple-resource problem. Philosopher  $P_i$  can be represented as a process  $P[i]$ , where the thinking phase corresponds to a noncritical activity that requires no (external) resources, while the eating phase corresponds to a critical activity that requires the chopsticks on the left and on the right of the philosopher. Philosopher  $P[i]$  competes with the philosopher on his right,  $P[i \oplus_M 1]$ , for the possession of his right chopstick  $c[i \oplus_M 1]$  and with the philosopher on his left,  $P[i \ominus_M 1]$ , for the

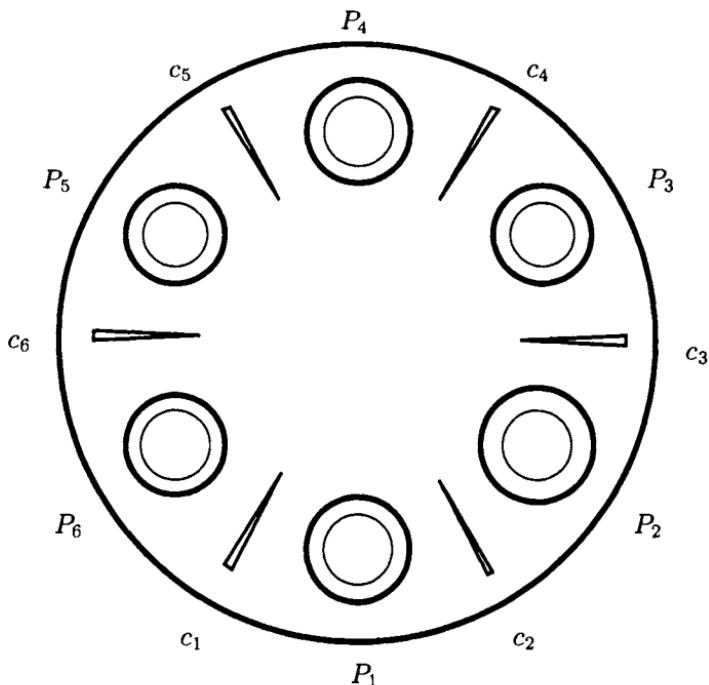


Fig. 2.14. Dining philosophers setup.

possession of his left chopstick  $c[i]$ . The operation  $i \ominus_M 1$  is defined by

$$i \ominus_M 1 = \begin{cases} i - 1 & \text{if } i > 1 \\ M & \text{if } i = 1. \end{cases}$$

### **Solution 1: Program DINE**

A simple solution to the dining philosophers problem is presented as program DINE in Fig. 2.15.

The solution presented in Fig. 2.15 uses an array of semaphores  $c$  to manage allocation of chopsticks. For philosopher  $j \in [1..M]$ ,  $c[j]$  represents the availability of the chopstick on his left. This means that  $c[j] = 1$  iff the chopstick is available. In a similar way,  $c[j \oplus_M 1]$  represents availability of the chopstick on his right. Addition has to be modulo  $M$  so that the chopstick to the right of  $P[M]$  is identified as  $c[M \oplus_M 1] = c[1]$ .

The protocol for the behavior of a philosopher process, as described in program DINE, is simple. As the process comes out of its noncritical section (thinking section) it first attempts to acquire the chopstick on its left by performing request  $c[j]$ . When this operation succeeds, the process attempts to acquire the

in  $M$ : integer where  $M \geq 2$   
 local  $c$  : array [1.. $M$ ] of integer where  $c = 1$

$\prod_{j=1}^M P[j] ::$   $\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } c[j] \\ \ell_3: \text{request } c[j \oplus_M 1] \\ \ell_4: \text{critical} \\ \ell_5: \text{release } c[j] \\ \ell_6: \text{release } c[j \oplus_M 1] \end{array} \right] \end{array} \right]$

Fig. 2.15. Program DINE: a simple solution to the dining philosophers problem.

chopstick on its right by performing **request**  $c[j \oplus_M 1]$ . When this operation succeeds, the process enters its critical section (eating section). On exit from the critical section, the process releases first its left chopstick and then its right chopstick.

The specification for the dining philosophers problem is not necessarily that of general mutual exclusion. In a system with  $M = 4$  philosophers, there is no objection to  $P_1$  and  $P_3$  eating at the same time. The relevant exclusion requirement stems from the indivisibility of the resources — the chopsticks — which can belong to only one philosopher at a time. Thus the specification for the system is given by the following invariant assertion

$$\psi: \forall j \in [1..M]: \neg(at_{-}\ell_4[j] \wedge at_{-}\ell_4[j \oplus_M 1]),$$

requiring mutual exclusion between every two *adjacent* philosophers, since they are the ones who share a chopstick. For obvious reasons, we refer to this property as *chopstick exclusion*.

Once this requirement is identified, it is not difficult to come up with inductive assertions similar to the ones we have described for other semaphore-using programs:

$$\varphi_0: \forall j \in [1..M]: c[j] \geq 0$$

$$\varphi_1: \forall j \in [1..M]: at_{-}\ell_{4..6}[j] + at_{-}\ell_{3..5}[j \oplus_M 1] + c[j \oplus_M 1] = 1.$$

Invariant  $\varphi_0$  simply states that, as a semaphore variable,  $c[j]$  can never become negative. Invariant  $\varphi_1$  states, for each chopstick  $c[j]$ ,  $j \in [1..M]$ , that the sum of  $c[j \oplus_M 1]$  and the characteristic predicates (interpreted arithmetically)  $at_{-}\ell_{4..6}[j]$

and  $at_{-\ell_3..5}[j \oplus_M 1]$  equal 1 at all times.

It is not difficult to ascertain that assertion  $\varphi_1$  is inductive. The relevant transitions are

- Transition  $\ell_3[j]$  increments  $at_{-\ell_4..6}[j]$ , while decrementing  $c[j \oplus_M 1]$ .
- Transition  $\ell_6[j]$  decrements  $at_{-\ell_4..6}[j]$ , while incrementing  $c[j \oplus_M 1]$ .
- Transition  $\ell_2[j \oplus_M 1]$  increments  $at_{-\ell_3..5}[j \oplus_M 1]$ , while decrementing  $c[j \oplus_M 1]$ .
- Transition  $\ell_5[j \oplus_M 1]$  decrements  $at_{-\ell_3..5}[j \oplus_M 1]$ , while incrementing  $c[j \oplus_M 1]$ .

Thus all transitions preserve the value of the sum  $at_{-\ell_4..6}[j] + at_{-\ell_3..5}[j \oplus_M 1] + c[j \oplus_M 1]$ .

It is obvious that  $\varphi_0$  and  $\varphi_1$  imply the required specification  $\psi$ . This is because the conjunction  $at_{-\ell_4}[j] \wedge at_{-\ell_4}[j \oplus_M 1]$  implies, by  $\varphi_1$ ,  $c[j \oplus_M 1] = -1$  which contradicts  $\varphi_0$ .

This concludes the proof of chopstick exclusion for program DINE.

## Improved Solutions: Breaking the Symmetry

While program DINE has been shown to satisfy the safety property of chopstick exclusion, it is deficient in satisfying the progress properties one may expect from such a program. Program DINE can reach a deadlock state from which no process can progress, which implies that all processes are denied access to their critical sections from this point on. We refer to such a situation as *starvation* because this is what will happen to philosophers who follow the DINE protocol and reach this deadlock state.

The deadlock state for program DINE is the state in which all processes are at location  $\ell_3$  in their respective programs. Due to invariant  $\varphi_1$ , all chopstick variables are 0 at such a state and therefore all  $\ell_3$  statements are disabled. In the philosophers' imaginary world, this corresponds to the situation that, by some incredible coincidence, all philosophers have grabbed their left chopstick and are trying to reach for their right chopstick which, alas, is strongly grasped in the hand of their right neighbor. Of course, none of the philosophers will put down his chopstick until he has eaten.

One might argue that such an amazing coincidence is highly unlikely, but since in our pessimistic view of the world we always consider the worst-case situation, we have to reject program DINE for not being starvation free.

Analyzing the deadlock situation, we can observe that, to a large extent, it is caused by symmetry. That is, it is due to the fact that the processes have acted in a symmetric manner, with each process picking its left chopstick, and thus they have created a perfectly symmetric deadlock situation.

A closely related observation is that program DINE does not follow our general recommendation for preventing deadlocks (see page 193). This recommendation suggested that all resources be ordered in some global order, and each process request the resources it needs in a sequence compatible with this global order. A natural global order for the chopstick semaphores is  $c[1], \dots, c[M]$ . Each of the processes  $P[j]$ , for  $j = 1, \dots, M-1$ , requests its chopsticks in a sequence compatible with this global order. It requests  $c[j]$  first and then asks for  $c[j \oplus_M 1] = c[j+1]$ . The exception is process  $P[M]$  which requests first  $c[M]$  and then asks for  $c[j \oplus_M 1] = c[1]$ .

There exist several starvation-free solutions to the dining philosophers problem that are based on breaking the symmetry. We consider two of them here.

### **Solution 2: A Contrary Philosopher**

The simplest way to break the symmetry is to have  $P[M]$  reverse the order of its requests. The resulting program DINE-CONTR is presented in Fig. 2.16. In this program, processes  $P[1], \dots, P[M-1]$  have symmetric programs but the program for process  $P[M]$  differs.

To show that program DINE-CONTR maintains chopstick exclusion, we can establish as in program DINE the following invariants:

- $$\begin{aligned}\varphi_0: \quad & \forall j \in [1..M]: c[j] \geq 0 \\ \varphi_1: \quad & \forall j \in [1..M-2]: at_{-}\ell_{4..6}[j] + at_{-}\ell_{3..5}[j+1] + c[j+1] = 1 \\ \varphi_2: \quad & at_{-}\ell_{4..6}[M-1] + at_{-}\ell_{4..6}[M] + c[M] = 1 \\ \varphi_3: \quad & at_{-}\ell_{3..5}[M] + at_{-}\ell_{3..5}[1] + c[1] = 1.\end{aligned}$$

Invariant  $\varphi_1$  expresses the fact that chopstick  $c[j+1]$  can be appropriated by either  $P[j]$  when it is in locations  $\ell_{4..6}$ , or by  $P[j+1]$  when it is in locations  $\ell_{3..5}$ , but not by both at the same time (because  $c[j+1] \geq 0$ ).

Note that  $\varphi_2$  and  $\varphi_3$  differ from the uniform pattern of  $\varphi_1$  due to the fact that  $P[M]$  requests its chopsticks in a reverse order. It can be shown that this solution is starvation free.

### **Solution 3: One Philosopher Excluded**

Program DINE-CONTR of Fig. 2.16 breaks the symmetry by syntactic means. That is, the program for  $P[M]$  is different from the programs for  $P[1], \dots, P[M-1]$ . The original dining philosophers problem formulation of Dijkstra asked for a syntactically symmetric solution.

A syntactically symmetric solution has the advantage that it uses identical components of only one kind. A company that specializes in providing dining

in  $M$ : integer where  $M \geq 2$   
 local  $c$  : array [1.. $M$ ] of integer where  $c = 1$

$\prod_{j=1}^{M-1} P[j] ::$	$\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } c[j] \\ \ell_3: \text{request } c[j+1] \\ \ell_4: \text{critical} \\ \ell_5: \text{release } c[j] \\ \ell_6: \text{release } c[j+1] \end{array} \right] \\ \dots \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \text{request } c[1] \\ \ell_3: \text{request } c[M] \\ \ell_4: \text{critical} \\ \ell_5: \text{release } c[1] \\ \ell_6: \text{release } c[M] \end{array} \right] \end{array} \right]$
-----------------------------	---

Fig. 2.16. Program DINE-CONTR: solution with one contrary philosopher.

philosopher systems of various sizes will have to keep quantities of only one type of component in stock. On the other hand, a company that bases its business on the protocol of program DINE-CONTR has to stock components of two types.

Consequently, we prefer solutions that are still syntactically symmetric but in which symmetry is broken dynamically. Such a solution is provided by program DINE-EXCL, presented in Fig. 2.17. This program uses an additional semaphore variable  $r$ , whose initial value is  $M-1$ .

Going back to the philosophers' world, this program corresponds to a change in the lifestyle of the philosophers. Philosophers no longer think at the dining table. We can envisage two rooms connected by a doorway, as shown in Fig. 2.18.

Philosophers do their thinking in the library. In order to eat they have to enter the dining hall and sit at their regular places, using chopsticks  $c[j]$  and  $c[j \oplus M 1]$  as before. However, the entrance to the dining hall is guarded by a

```

in      M: integer where M ≥ 2
local c : array [1..M] integer where c = 1
      r : integer where r = M - 1

```

$\prod_{j=1}^M P[j] ::$	$\ell_0:$ loop forever do <table border="0" style="margin-left: 20px;"> <tr><td><math>\ell_1:</math> noncritical</td></tr> <tr><td><math>\ell_2:</math> request <math>r</math></td></tr> <tr><td><math>\ell_3:</math> request <math>c[j]</math></td></tr> <tr><td><math>\ell_4:</math> request <math>c[j \oplus_M 1]</math></td></tr> <tr><td><math>\ell_5:</math> critical</td></tr> <tr><td><math>\ell_6:</math> release <math>c[j]</math></td></tr> <tr><td><math>\ell_7:</math> release <math>c[j \oplus_M 1]</math></td></tr> <tr><td><math>\ell_8:</math> release <math>r</math></td></tr> </table>	$\ell_1:$ noncritical	$\ell_2:$ request $r$	$\ell_3:$ request $c[j]$	$\ell_4:$ request $c[j \oplus_M 1]$	$\ell_5:$ critical	$\ell_6:$ release $c[j]$	$\ell_7:$ release $c[j \oplus_M 1]$	$\ell_8:$ release $r$
$\ell_1:$ noncritical									
$\ell_2:$ request $r$									
$\ell_3:$ request $c[j]$									
$\ell_4:$ request $c[j \oplus_M 1]$									
$\ell_5:$ critical									
$\ell_6:$ release $c[j]$									
$\ell_7:$ release $c[j \oplus_M 1]$									
$\ell_8:$ release $r$									

Fig. 2.17. Program DINE-EXCL: one philosopher excluded.

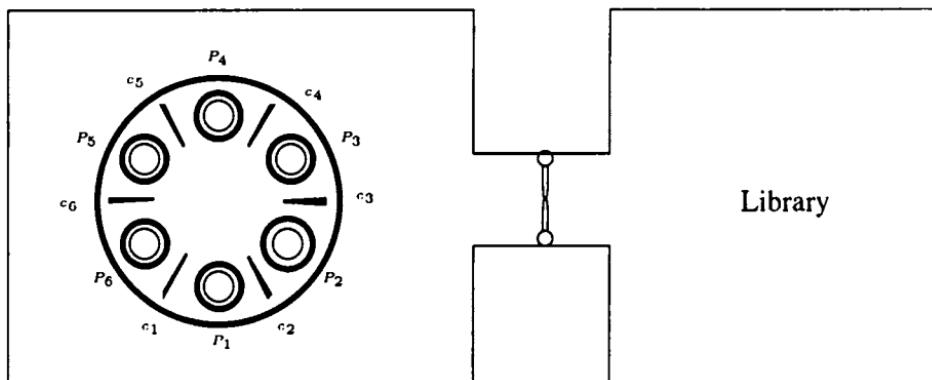


Fig. 2.18. Two-room philosophers' world.

stern doorkeeper who never admits more than  $M-1$  philosophers to the dining hall. When a philosopher finishes eating, he returns to the library.

The doorkeeper is represented in program DINE-EXCL by the semaphore  $r$ . Each process has to go through the **request  $r$**  statement before it can start competing for the chopsticks.

As shown in the PROGRESS book, this solution is also starvation free and

guarantees that any process reaching location  $\ell_2$  eventually reaches its critical section at location  $\ell_5$ .

One can argue that the success of this program is also due to symmetry breaking. Assume that all philosophers get hungry at the same time, leaving location  $\ell_1$  to arrive at the doorway at  $\ell_2$ . At most  $M - 1$  will be admitted to  $\ell_3$  and a lonely philosopher will remain stuck at  $\ell_2$ . When only  $M - 1$  philosophers are competing for the  $M$  chopsticks, symmetry is broken and at least one philosopher will get to eat. However, unlike program DINE-CONTR, symmetry is not broken by syntactic means, singling out a fixed process that behaves differently than its brothers. The determination of the philosopher who remains behind at  $\ell_2$  is dynamic in program DINE-EXCL. This means that, during different rounds of competition, different processes may have to wait at  $\ell_2$ .

While the addition of semaphore  $r$  improves the progress properties of the proposed solution, it does not contribute to its safety properties. Chopstick exclusion is still ensured by the following invariants which are fully analogous to the corresponding invariants for program DINE:

$$\varphi_0: \forall j \in [1..M]: c[j] \geq 0$$

$$\varphi_1: \forall j \in [1..M]: at_{-\ell_5..7}[j] + at_{-\ell_4..6}[j \oplus_M 1] + c[j \oplus_M 1] = 1.$$

These two assertions are sufficient to establish chopstick exclusion.

In preparation for the proof of progress properties of program DINE-EXCL conducted in the PROGRESS book, we derive the following two invariants that characterize the behavior of variable  $r$  and its connection to the location of processes:

$$\varphi_2: r \geq 0$$

$$\varphi_3: N_{3..8} + r = M - 1.$$

Together, invariants  $\varphi_2$  and  $\varphi_3$  state that there cannot be more than  $M - 1$  processes in the location range  $\ell_{3..8}$ .

In **Problems 2.6** and **2.7**, the reader is requested to verify programs that detect termination in a system of processes arranged in a ring of synchronous and asynchronous channels, respectively. In **Problem 2.8**, the reader is requested to verify a program that computes a fixpoint of a set of equations in a parallel fashion.

## 2.4 Constructing Linear Invariants

There is a restricted set of invariants that can be constructed algorithmically for a given program. Their construction is bottom-up since they are independent of any goal assertions.

An integer data variable  $y$  is called *linear* in a program  $P$  if the effect of any transition of  $P$  on  $y$  can be expressed as  $y' = y + C$ , where  $C$  is some integer constant, possibly  $C = 0$ . The constant  $C$  may vary from transition to transition. For example, every semaphore variable  $y$  is linear since the only transitions modifying it either subtract 1 or add 1 to  $y$ .

Let  $\mathcal{L}$  be the set of locations of the program, and let  $y_1, \dots, y_r$  be all the linear variables of  $P$ . We are interested in invariants of the form

$$\sum_{i=1}^r a_i \cdot y_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot at_\ell = K,$$

where  $a_i$ ,  $b_\ell$ , and  $K$  are integer constants.

We refer to an invariant of this form as a *linear invariant*, and to the terms

$$\sum_{i=1}^r a_i \cdot y_i, \quad \sum_{\ell \in \mathcal{L}} b_\ell \cdot at_\ell, \quad \text{and} \quad K,$$

as the *body*, *compensation expression*, and *right constant* of the invariant, respectively.

**Example** Consider program DOUBLE of Fig. 2.19 (see also Fig. 1.24, page 151).

local  $y$ : integer where  $y = 0$

$$\left[ \begin{array}{l} \ell_0: y := y + 1 \\ \ell_1: \end{array} \right] \parallel \left[ \begin{array}{l} m_0: y := y + 1 \\ m_1: \end{array} \right]$$

Fig. 2.19. Program DOUBLE (double incrementation).

It is obvious that variable  $y$  is linear in this program. The assertion

$$y + (at_{\ell_0} + at_{m_0}) = 2$$

is a linear invariant with coefficients  $a_y = b_{\ell_0} = b_{m_0} = 1$ . The body of the invariant is  $y$ , the compensation expression is  $at_{\ell_0} + at_{m_0}$  and the right constant  $K$  is 2.

## Increments

For simplicity, we restrict ourselves to programs in which the only cooperation statement is the body of the program, and each linear variable  $y$  is prescribed

an initial value  $y^0$ . In the absence of nested parallel statements, the transition relation of a transition  $\tau$  refers to control only through a conjunct  $\text{move}(\ell_i, \ell_j)$ . We say that  $\tau$  leads from  $\ell_i$  to  $\ell_j$ .

Let  $\tau$  be a transition and  $y$  a linear variable. We define the *increment to  $y$  by  $\tau$* , denoted  $\Delta(y, \tau)$ , as follows:

$$\Delta(y, \tau) = C \quad \text{if } \rho_\tau \text{ implies } y' = y + C.$$

Thus, the increment  $\Delta(y, \tau)$  is the amount added to  $y$  by the execution of  $\tau$ , i.e.,

$$\rho_\tau \text{ implies } y' = y + \Delta(y, \tau).$$

For an arbitrary label  $\ell$  and transition  $\tau$ , leading from  $\ell_i$  to  $\ell_j$ , we define the increment of the control predicate  $\text{at-}\ell$  by  $\tau$  as

$$\Delta(\text{at-}\ell, \tau) = \begin{cases} +1 & \text{if } \ell \sim \ell_j \\ -1 & \text{if } \ell \sim \ell_i \\ 0 & \text{otherwise.} \end{cases}$$

Thus,

$$\rho_\tau \text{ implies } \text{at}'_\ell = \text{at-}\ell + \Delta(\text{at-}\ell, \tau).$$

We can extend the notion of increment to linear expressions over linear variables such as  $a_1 \cdot y_1 + \dots + a_k \cdot y_k$ , where  $y_1, \dots, y_k$  are linear variables, and also over  $\text{at-}\ell$  predicates by defining

$$\Delta(c, \tau) = 0 \text{ for any constant } c, \text{ and}$$

$$\Delta(a_1 \cdot t_1 + \dots + a_k \cdot t_k, \tau) = a_1 \cdot \Delta(t_1, \tau) + \dots + a_k \cdot \Delta(t_k, \tau).$$

## Equations

Assume that the program consists of the processes  $S_1, \dots, S_m$  with initial locations  $\ell_0^1, \dots, \ell_0^m$ . We wish to construct an invariant of the form

$$\varphi: \sum_{i=1}^r a_i \cdot y_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \text{at-}\ell = K.$$

The following set of equations in the unknowns  $a_i$ ,  $b_\ell$ , and  $K$  is a necessary and sufficient condition for the invariance of assertion  $\varphi$ :

$$(T) \quad \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau) + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \Delta(\text{at-}\ell, \tau) = 0 \quad \text{for every transition } \tau$$

$$(I) \quad \sum_{i=1}^r a_i \cdot y_i^0 + (b_{\ell_0^1} + \dots + b_{\ell_0^m}) = K.$$

As  $K$  appears only in the initiality equation (I), the recommended method of

solution is to first solve the transition equations (T) for  $a_1, \dots, a_r$ , and  $b_\ell$ ,  $\ell \in \mathcal{L}$ , and then use equation (I) to derive a value for  $K$ , given the solution values for the  $a_i$ 's and  $b_\ell$ 's.

**Example** Consider program DOUBLE of Fig. 2.19. Its set of locations is  $\mathcal{L} = \{\ell_0, \ell_1, m_0, m_1\}$ . This program has the linear variable  $y$ . Consequently, we search for an invariant of the form

$$\varphi: a \cdot y + b_{\ell_0} \cdot at_{-\ell_0} + b_{\ell_1} \cdot at_{-\ell_1} + b_{m_0} \cdot at_{-m_0} + b_{m_1} \cdot at_{-m_1} = K.$$

We compute the following increments

$$\begin{array}{lll} \Delta(y, \ell_0) = 1, & \Delta(at_{-\ell_0}, \ell_0) = -1, & \Delta(at_{-\ell_1}, \ell_0) = +1, \\ & \Delta(at_{-m_0}, \ell_0) = 0, & \Delta(at_{-m_1}, \ell_0) = 0, \\ \Delta(y, m_0) = 1, & \Delta(at_{-\ell_0}, m_0) = 0, & \Delta(at_{-\ell_1}, m_0) = 0, \\ & \Delta(at_{-m_0}, m_0) = -1, & \Delta(at_{-m_1}, m_0) = +1. \end{array}$$

Equations (T) assume the following form

$$\begin{array}{lll} a - b_{\ell_0} + b_{\ell_1} = 0 & & \text{(for transition } \ell_0) \\ a - b_{m_0} + b_{m_1} = 0 & & \text{(for transition } m_0) \end{array}$$

Equation (I) is

$$b_{\ell_0} + b_{m_0} = K,$$

since the initial value of  $y$  is 0 and  $\ell_1, m_1$  are not initial locations.

Three possible solutions to (T) and then (I) are

$$\begin{array}{lll} S_1: b_{\ell_0} = b_{\ell_1} = K = 1, & a = b_{m_0} = b_{m_1} = 0 \\ S_2: b_{m_0} = b_{m_1} = K = 1, & a = b_{\ell_0} = b_{\ell_1} = 0 \\ S_3: a = b_{\ell_0} = b_{m_0} = 1, & K = 2, & b_{\ell_1} = b_{m_1} = 0. \end{array}$$

It can be shown that these solutions form a *basis* for all solutions of the combined system (T)+(I). This means that any solution is expressible as a linear combination of the above three solutions.

The assertions corresponding to these solutions are

$$\begin{array}{l} \varphi_1: at_{-\ell_0} + at_{-\ell_1} = 1 \\ \varphi_2: at_{-m_0} + at_{-m_1} = 1 \\ \varphi_3: y + at_{-\ell_0} + at_{-m_0} = 2. \end{array}$$

Indeed, these three assertions are invariant over program DOUBLE. ■

## Correctness of the Construction

The correctness of the construction is stated by the following two claims:

**Claim 2.1** ((T)+(I) are sufficient)

Any solution of equations (T)+(I) yields a  $P$ -invariant assertion.

**Justification** To substantiate the claim, let

$$\varphi: \sum_{i=1}^r a_i \cdot y_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot at_\ell = K$$

be an assertion derived from a solution to equations (T)+(I). We will show that  $\varphi$  satisfies the premises of rule INV-B.

Premise B1 requires showing  $\Theta \rightarrow \varphi$ . Since  $\Theta$  implies  $y_i = y_i^0$ , for  $i = 1, \dots, r$ ,  $at_\ell = \tau = 1$  for  $\ell \in \{\ell_0^1, \dots, \ell_0^m\}$  and  $at_\ell = \text{F} = 0$  for  $\ell \notin \{\ell_0^1, \dots, \ell_0^m\}$ , it is sufficient to show

$$\sum_{i=1}^r a_i \cdot y_i^0 + (b_{\ell_0^1} + \dots + b_{\ell_0^m}) = K,$$

which is given by equation (I).

Premise B2 requires showing  $(\rho_\tau \wedge \varphi) \rightarrow \varphi'$  for every transition  $\tau$ , that is,

$$\left( \rho_\tau \wedge \underbrace{\sum_{i=1}^r a_i \cdot y_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot at_\ell = K}_{\varphi} \right) \rightarrow \underbrace{\sum_{i=1}^r a_i \cdot y'_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot at'_\ell = K}_{\varphi'}$$

which can be rewritten as

$$\rho_\tau \rightarrow \left( \sum_{i=1}^r a_i \cdot (y'_i - y_i) + \sum_{\ell \in \mathcal{L}} b_\ell \cdot (at'_\ell - at_\ell) = 0 \right).$$

As observed when defining the increments of variables and control predicates,  $\rho_\tau$  implies  $y'_i = y_i + \Delta(y_i, \tau)$  for  $i = 1, \dots, r$ , and  $at'_\ell = at_\ell + \Delta(at_\ell, \tau)$  for  $\ell \in \mathcal{L}$ . Thus, it is sufficient to prove

$$\sum_{i=1}^r a_i \cdot \Delta(y_i, \tau) + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \Delta(at_\ell, \tau) = 0,$$

which is given by (T).

It follows by rule INV-B, that  $\varphi$  is inductive and, therefore, invariant.

**Claim 2.2** ((T)+(I) are necessary)

The coefficients  $a_i$ ,  $i = 1, \dots, r$ , and  $b_\ell$ ,  $\ell \in \mathcal{L}$ , and the right constant  $K$  of a  $P$ -invariant assertion

$$\varphi: \sum_{i=1}^r a_i \cdot y_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot at_\ell = K$$

satisfy equations (T)+(I).

**Justification** As  $\varphi$  is  $P$ -invariant, it holds on the initial state of program  $P$  in which  $y_1 = y_1^0, \dots, y_r = y_r^0$  and  $\pi = \{\ell_0^1, \dots, \ell_0^m\}$ . Substituting these values in  $\varphi$ , we obtain

$$\sum_{i=1}^r a_i \cdot y_i^0 + (b_{\ell_0^1} + \dots + b_{\ell_0^m}) = K$$

which is precisely equation (I).

Let  $\tau$  be a transition of  $P$ ,  $s$  the  $P$ -accessible state on which  $\tau$  is enabled, and  $s'$  the  $\tau$ -successor of  $s$ . For each  $i \in [1..r]$  and each  $\ell \in \mathcal{L}$ , let  $\eta_i$  denote  $s[y_i]$ , the value assigned to  $y_i$  by state  $s$ , and let  $\lambda_\ell$  denote  $s[at_\ell]$ , the value of  $at_\ell$  when evaluated over  $s[\pi]$ . Similarly, let  $\eta'_i = s'[y_i]$  and  $\lambda'_\ell = s'[at_\ell]$ . Evaluating  $\varphi$  first in  $s$  and then in  $s'$ , we obtain the following two equalities:

$$\begin{aligned} \sum_{i=1}^r a_i \cdot \eta_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \lambda_\ell &= K \\ \sum_{i=1}^r a_i \cdot \eta'_i + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \lambda'_\ell &= K. \end{aligned}$$

Subtracting the first equality from the second, we obtain

$$\sum_{i=1}^r a_i \cdot (\eta'_i - \eta_i) + \sum_{\ell \in \mathcal{L}} b_\ell \cdot (\lambda'_\ell - \lambda_\ell) = 0.$$

Using the definitions of  $\Delta(y_i, \tau)$  and  $\Delta(at_\ell, \tau)$ , this equality can be written as

$$\sum_{i=1}^r a_i \cdot \Delta(y_i, \tau) + \sum_{\ell \in \mathcal{L}} b_\ell \cdot \Delta(at_\ell, \tau) = 0$$

which is exactly equation (T) for the particular  $\tau$  considered. ■

## Cyclic Programs

While equations (T)+(I) are sufficient to construct a large class of linear invariants, they are not always convenient to solve in practice, because the number of unknowns is typically large. We now present a more structured approach to the solution of those equations, one that proceeds in three phases. The first phase identifies a solution for the unknowns  $a_1, \dots, a_r$ . The second phase uses the values obtained for  $a_1, \dots, a_r$  to calculate values for  $b_\ell$ ,  $\ell \in \mathcal{L}$ . The last phase computes  $K$ .

The more structured method is applicable to *cyclic programs*. These are programs in which every process  $P_j$  consists of an infinite loop of the form

$\ell_0^j: \text{loop forever do } S_j.$

A (*simple*) *cycle* in a program is a sequence of pairwise disjoint locations  $C: \ell_1, \dots, \ell_k$  and corresponding transitions  $\tau_1, \dots, \tau_k$  such that  $\tau_i$  leads from  $\ell_i$  to  $\ell_{i+1}$ , for  $i = 1, \dots, k-1$ , and  $\tau_k$  leads from  $\ell_k$  to  $\ell_1$ . For an expression  $e$  that can be either a linear variable or a control predicate, we define the notion of an *increment along a cycle*  $C: \ell_1, \dots, \ell_k$ , given by

$$\Delta(e, C) = \Delta(e, \tau_1) + \dots + \Delta(e, \tau_k).$$

### Phase 1: Bodies

Consider the equations of (T) that correspond to the transitions  $\tau_1, \dots, \tau_k$  which constitute the cycle  $C$ . They can be written as follows:

$$\begin{aligned} \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau_1) - b_{\ell_1} + b_{\ell_2} &= 0 \\ \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau_2) - b_{\ell_2} + b_{\ell_3} &= 0 \\ &\vdots && \vdots \\ \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau_k) + b_{\ell_1} - b_{\ell_k} &= 0. \end{aligned}$$

If we sum all the equations together, we see that the parts dealing with  $b_{\ell_1}, \dots, b_{\ell_k}$  cancel each other, and we obtain the single equation

$$\sum_{i=1}^r a_i (\Delta(y_i, \tau_1) + \dots + \Delta(y_i, \tau_k)) = \sum_{i=1}^r 0 = 0,$$

which can also be written as

$$\sum_{i=1}^r a_i \cdot \Delta(y_i, C) = 0.$$

Such an equation is formed for each cycle  $C$  in the program.

### Example (NONSENSE)

Consider program NONSENSE presented in Fig. 2.20. While not representing any interesting algorithm, this program demonstrates a richer class of control paths than other previously shown examples. The linear variables are  $x$  and  $y$  ( $z$  is not a linear variable due to the assignment at  $\ell_4$ ). Note that

$$\ell_2 \sim_L \ell_2^a \sim_L \ell_2^b$$

and all three labels refer to the same location  $[\ell_2]$ .

The program has three simple cycles

$$C_1: \ell_0, \ell_1, \ell_2^a, \ell_3, \ell_4, \ell_5^F, \ell_{10}$$

$$C_2: \ell_0, \ell_1, \ell_2^b, \ell_4, \ell_5^F, \ell_{10}$$

$$C_3: \ell_5^T, \ell_6, \ell_7, \ell_8, \ell_9.$$

```
local x, y, z: integer where x = y = z = 0
```

```
 $\ell_0$ : loop forever do
```

$\ell_1: x := x + 1$ $\ell_2:$ $\left[ \begin{array}{l} \ell_2^a: x := x - 1; \ell_3: y := y + 2 \\ \text{or} \\ \ell_2^b: z := z + 1 \end{array} \right]$ $\ell_4: z :=  x  +  y $ $\ell_5: \text{while } z > 0 \text{ do}$	$\left[ \begin{array}{l} \ell_6: y := y + 1 \\ \ell_7: x := x - 1 \\ \ell_8: y := y + 1 \\ \ell_9: z := z - 1 \end{array} \right]$ $\ell_{10}: y := y - 2$
---	---

Fig. 2.20. Program NONSENSE.

Note that cycles  $C_1$  and  $C_2$  only differ in that  $C_1$  proceeds from  $\ell_1$  to  $\ell_4$  via  $\ell_2^a$

and  $\ell_3$ , while  $C_2$  takes transition  $\ell_2^b$ .

In computing increments to  $x$ , we should only consider transitions  $\ell_1$ ,  $\ell_2^a$ , and  $\ell_7$ , with contributions +1, -1, and -1, respectively. Transitions that contribute to the increments of  $y$  are  $\ell_3$ ,  $\ell_6$ ,  $\ell_8$ , and  $\ell_{10}$ , with contributions +2, +1, +1, and -2, respectively. Consequently, the increments along the cycles are given by

$$\begin{array}{lll} \Delta(x, C_1) = +1 - 1 = 0 & \Delta(y, C_1) = +2 - 2 = 0 \\ \Delta(x, C_2) = +1 = 1 & \Delta(y, C_2) = -2 = -2 \\ \Delta(x, C_3) = -1 = -1 & \Delta(y, C_3) = +1 + 1 = 2. \end{array}$$

It follows that the cycle equations are

$$\begin{aligned} 0 \cdot a_x + 0 \cdot a_y &= 0 & (\text{for } C_1) \\ a_x - 2 \cdot a_y &= 0 & (\text{for } C_2) \\ -a_x + 2 \cdot a_y &= 0 & (\text{for } C_3). \end{aligned}$$

A basis for the solutions of these equations is given by

$$a_x = 2, \quad a_y = 1.$$

Thus, we have identified the invariant body

$$2x + y.$$

### Example (SEM-3)

As another running example, consider program SEM-3 of Fig. 2.21 which achieves mutual exclusion by the use of three semaphore variables  $y_1$ ,  $y_2$ , and  $y_3$ .

Obviously, the linear variables are  $y_1$ ,  $y_2$ , and  $y_3$ . The program has three cycles, corresponding to processes  $P_1$ ,  $P_2$ , and  $P_3$ , respectively. The increments of  $y_1$ ,  $y_2$ ,  $y_3$  along these three cycles are given by

$$\begin{array}{lll} \Delta(y_1, C_1) = -1, & \Delta(y_2, C_1) = 1, & \Delta(y_3, C_1) = 0 \\ \Delta(y_1, C_2) = 0, & \Delta(y_2, C_2) = -1, & \Delta(y_3, C_2) = 1 \\ \Delta(y_1, C_3) = 1, & \Delta(y_2, C_3) = 0, & \Delta(y_3, C_3) = -1. \end{array}$$

Considering an invariant body of the form  $a_1 \cdot y_1 + a_2 \cdot y_2 + a_3 \cdot y_3$ , we obtain the following three cycle equations:

$$\begin{aligned} -a_1 + a_2 &= 0 \\ -a_2 + a_3 &= 0 \\ a_1 - a_3 &= 0. \end{aligned}$$

One solution of these equations is given by

**local**  $y_1, y_2, y_3$ : **integer** **where**  $y_1 = 1, y_2 = y_3 = 0$

$$\begin{array}{ll}
 P_1 :: & \left[ \begin{array}{l} \ell_0^1: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1^1: \text{noncritical} \\ \ell_2^1: \text{request } y_1 \\ \ell_3^1: \text{critical} \\ \ell_4^1: \text{release } y_2 \end{array} \right] \end{array} \right] \\
 & || \\
 P_2 :: & \left[ \begin{array}{l} \ell_0^2: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1^2: \text{noncritical} \\ \ell_2^2: \text{request } y_2 \\ \ell_3^2: \text{critical} \\ \ell_4^2: \text{release } y_3 \end{array} \right] \end{array} \right] \\
 & || \\
 P_3 :: & \left[ \begin{array}{l} \ell_0^3: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1^3: \text{noncritical} \\ \ell_2^3: \text{request } y_3 \\ \ell_3^3: \text{critical} \\ \ell_4^3: \text{release } y_1 \end{array} \right] \end{array} \right]
 \end{array}$$

Fig. 2.21. Program SEM-3 (three semaphores).

$$a_1 = a_2 = a_3 = 1.$$

It is not difficult to see that this is a basis for all the solutions of the cycle equations. That is, any other solution is a multiple of this solution by some constant. ■

## Phase 2: Compensation Expressions

In phase 2, we use equations (T) to derive values for  $b_\ell$ ,  $\ell \in \mathcal{L}$ , given a solution for  $a_1, \dots, a_r$ . These values can be derived separately for each process. Consider a particular process  $P_j$  and let its locations be  $\ell_0, \ell_1, \dots$ . We assume that for every location  $\ell_n$ ,  $n > 0$ , there exists at least one location  $\ell_k$ ,  $k < n$ , and some

transition  $\tau$  leading from  $\ell_k$  to  $\ell_n$ . We determine the values of  $b_{\ell_0}, b_{\ell_1}, \dots$  in order of increasing label indices:

- $b_{\ell_0} = 0$ .
- For  $n > 0$ , let  $\tau$  be a transition leading from  $\ell_k$  to  $\ell_n$ ,  $k < n$ . Assume that we have already determined the values of  $b_{\ell_i}$  for all  $i < n$ , including  $b_{\ell_k}$ . Equations (T) contain the equation (for transition  $\tau$ )

$$\sum_{i=1}^r a_i \cdot \Delta(y_i, \tau) - b_{\ell_k} + b_{\ell_n} = 0,$$

from which we can calculate

$$b_{\ell_n} = b_{\ell_k} - \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau).$$

This derivation is repeated for each process  $P_j$ .

In the computation of  $b_{\ell_n}$  for  $n > 0$ , it may happen that there exist more than one  $k < n$  and transition  $\tau$  leading from  $\ell_k$  to  $\ell_n$ . In such cases, it appears that we may have two different ways to compute  $b_{\ell_n}$ . However, it can be shown that if  $a_1, \dots, a_r$  satisfy the body equations, then all possible computations of  $b_{\ell_n}$  yield the same value.

### Example (NONSENSE)

Continuing with the construction of a linear invariant for program NONSENSE, we determine the values of  $b_{\ell_0}, \dots, b_{\ell_{10}}$  corresponding to the body  $2x + y$ .

$$b_{\ell_0} = 0$$

$$b_{\ell_1} = 0 - \Delta(2x + y, \ell_0) = 0$$

$$b_{\ell_2} = 0 - \Delta(2x + y, \ell_1) = 0 - 2 = -2$$

$$b_{\ell_3} = -2 - \Delta(2x + y, \ell_2) = -2 + 2 = 0$$

$$b_{\ell_4} = \left\{ \begin{array}{l} b_{\ell_3} - \Delta(2x + y, \ell_3) = 0 - 2 \\ b_{\ell_2} - \Delta(2x + y, \ell_2^b) = -2 - 0 \end{array} \right\} = -2$$

$$b_{\ell_5} = -2 - \Delta(2x + y, \ell_4) = -2 - 0 = -2$$

$$b_{\ell_6} = -2 - \Delta(2x + y, \ell_5^T) = -2 - 0 = -2$$

$$b_{\ell_7} = -2 - \Delta(2x + y, \ell_6) = -2 - 1 = -3$$

$$b_{\ell_8} = -3 - \Delta(2x + y, \ell_7) = -3 + 2 = -1$$

$$b_{\ell_9} = -1 - \Delta(2x + y, \ell_8) = -1 - 1 = -2$$

$$b_{\ell_{10}} = b_{\ell_5} - \Delta(2x + y, \ell_5^F) = -2 - 0 = -2.$$

The compensation expression is therefore given by

$$- 2 \cdot at\_l_2 - 2 \cdot at\_l_4 - 2 \cdot at\_l_5 - 2 \cdot at\_l_6 - \\ 3 \cdot at\_l_7 - at\_l_8 - 2 \cdot at\_l_9 - 2 \cdot at\_l_{10},$$

which can be written as

$$- 2 \cdot at\_l_{2,4..6,9,10} - 3 \cdot at\_l_7 - at\_l_8.$$

Thus the left-hand side of the constructed linear assertion is

$$2 \cdot x + y - 2 \cdot at\_l_{2,4..6,9,10} - 3 \cdot at\_l_7 - at\_l_8.$$

Note that there are two distinct ways to compute  $b_{l_4}$ . However, as previously explained, both computations yield the same value,  $-2$ . This is also true for coefficient  $b_{l_5}$ . ■

### Example (SEM-3)

Returning to program SEM-3, we previously obtained the single invariant body, given by

$$a_1 = a_2 = a_3.$$

Computing the coefficients  $b_\ell$ , we obtain

$$\begin{aligned} b_{\ell_0^1} &= b_{\ell_1^1} = b_{\ell_2^1} = b_{\ell_0^2} = b_{\ell_1^2} = b_{\ell_2^2} = b_{\ell_0^3} = b_{\ell_1^3} = b_{\ell_2^3} = 0 \\ b_{\ell_3^1} &= b_{\ell_4^1} = b_{\ell_3^2} = b_{\ell_4^2} = b_{\ell_3^3} = b_{\ell_4^3} = 1. \end{aligned}$$

This generates the following left-hand side of an invariant:

$$y_1 + y_2 + y_3 + at\_l_{3,4}^1 + at\_l_{3,4}^2 + at\_l_{3,4}^3. ■$$

### Phase 3: Right Constants

In the last phase, we compute the right-hand side constant  $K$  by

$$K = \sum_{i=1}^r a_i \cdot y_i^0.$$

Note that, since all coefficients  $b_{\ell_0^j} = 0$ , the initial value of the compensation expression is 0 and does not contribute to the value of  $K$ .

### Example (NONSENSE)

We resume the construction of a linear invariant for program NONSENSE. To determine the right-hand side of the assertion, we substitute the initial values of  $x$ ,  $y$  and obtain the following value of  $K$ :

$$K = \underbrace{0}_x + 2 \cdot \underbrace{0}_y = 0.$$

This leads to the assertion

$$\psi: 2 \cdot x + y - 2 \cdot at\_l_{2,4..6,9,10} - 3 \cdot at\_l_7 - at\_l_8 = 0,$$

which is invariant over program NONSENSE. ■

### Example (SEM-3)

Returning to the example of program SEM-3, we previously derived the following left-hand side of an invariant:

$$y_1 + y_2 + y_3 + at\_l_{3,4}^1 + at\_l_{3,4}^2 + at\_l_{3,4}^3.$$

Since the initial value of  $y_1 + y_2 + y_3$  is 1, we obtain the invariant assertion

$$y_1 + y_2 + y_3 + at\_l_{3,4}^1 + at\_l_{3,4}^2 + at\_l_{3,4}^3 = 1. ■$$

## Summary

We may summarize the set of equations used in the method for cyclic programs as follows.

- *Bodies*

For every cycle  $C$ ,

$$\sum_{i=1}^r a_i \cdot \Delta(y_i, C) = 0.$$

- *Compensation expressions*

$$b_{\ell_0^1} = \dots = b_{\ell_0^m} = 0.$$

For every  $\ell_n \notin \{\ell_0^1, \dots, \ell_0^m\}$  and some  $\tau$  leading from  $\ell_k$ ,  $k < n$ , to  $\ell_n$ ,

$$b_{\ell_n} = b_{\ell_k} - \sum_{i=1}^r a_i \cdot \Delta(y_i, \tau).$$

- *Right constants*

$$K = \sum_{i=1}^r a_i \cdot y_i^0.$$

We refer to this set of equations as the *cyclic equations* (CE).

The description of the calculation in Phase 2 and the corresponding equations contain several elements that may seem arbitrary. The first such element is taking all  $b_{\ell_0^j}$  to be 0. Another element that seems arbitrary is that the value of  $b_{\ell_n}$  is determined based on *some* transition leading from some  $\ell_k$ ,  $k < n$ , to  $\ell_n$ . In principle, there may be additional transitions that lead to  $\ell_n$ . Two such cases were illustrated in the example of program NONSENSE, where the values of  $b_{\ell_4}$  and  $b_{\ell_5}$  were computed in two different ways that yielded the same value. This is no

accident. If the body is determined by the cycle equations, then any determination of the  $b_{\ell_n}$  will always yield the same value.

The correspondence between equations (CE) and the combined set (T)+(I) is established by the following claim:

**Claim 2.3** (correspondence between (CE), (T)+(I))

Any linear assertion obtained as a solution to the cyclic equations (CE) is a solution of equations (T)+(I).

In **Problem 2.9**, we ask the reader to prove this claim, which establishes that the set of solutions to the cyclic equations is a subset of the set of solutions of (T)+(I). Thus, every assertion satisfying the cyclic equations is  $P$ -invariant.

### Limitations

Are there perhaps invariant assertions obtained as a solution to (T)+(I) which do not satisfy the cyclic equations? The answer is “yes.” By requiring all  $b_0^j$  to be 0, for each process  $P_j$ ,  $j = 1, \dots, m$ , we rule out some linear invariants that solve (T)+(I). For example, for every  $j = 1, \dots, m$ , let process  $P_j$  have the form

$$\ell_0^j: \text{loop forever do } \underbrace{\ell_1^j:S_1^j; \dots; \ell_t^j:S_t^j}_{S_j} .$$

Then the assertion

$$\chi_j: at\_l_0^j + \dots + at\_l_t^j = 1$$

is  $P$ -invariant and satisfies (T)+(I) but not (CE). Thus, for program NONSENSE, the invariant assertion

$$at\_l_{0..10} = 1$$

cannot be obtained as a solution to (CE). Note that we are allowed to write  $at\_l_{0..10}$  instead of the sum  $at\_l_0 + \dots + at\_l_{10}$  only because these 11 locations are pairwise conflicting.

However, these  $m$  assertions  $\chi_j$  are the only ones missed by equations (CE), as stated by the following claim.

**Claim 2.4** (linear combination)

Any solution of (T)+(I) can be obtained as a linear combination of a solution of (CE) and the assertions  $\chi_1, \dots, \chi_m$ .

In **Problem 2.10**, we ask the reader to prove this claim.

As an example, the linear assertion

$$2 \cdot x + y + 2 \cdot at\_l_0 + 2 \cdot at\_l_3 - at\_l_7 + at\_l_8 = 2$$

satisfies equations (T)+(I) for program NONSENSE but not (CE). On the other hand, it can be expressed as a linear combination of assertion  $\psi$  that satisfies (CE) and the assertion  $at_{-\ell_{0..10}} = 1$ . In fact, the presented linear assertion is obtained by adding together  $\psi$  and the equality  $2 \cdot at_{-\ell_0} + \dots + 2 \cdot at_{-\ell_{10}} = 2$  obtained by multiplying both sides of  $at_{-\ell_{0..10}} = 1$  by 2.

As a final observation, it is important to realize that neither (CE), nor the more general (T)+(I), are guaranteed to provide all linear invariants of the program. In particular, these approaches completely ignore the tests appearing in the program. Thus, if  $P$  is a program containing a test, say  $x > 0$  in one of its statements, and  $P'$  is obtained from  $P$  by replacing  $x > 0$  by its negation  $x \leq 0$ , then both (T)+(I) and (CE) for these two programs will consist of precisely the same equations. Consequently, linear invariants that depend on the particular tests appearing in the program cannot be constructed by (T)+(I) or (CE).

**Example** Consider program INSENSITIVE, presented in Fig. 2.22.

```

local x: integer where x = 0

l0: loop forever do
    [l1: x := x + 1
     l2: while x > 0 do
         [l3: x := x - 1
          l4: x := x + 1]
    ]
l5:

```

Fig. 2.22. Program INSENSITIVE.

The assertion

$$\psi: x - at_{-\ell_{2..3}} = 0$$

is a linear invariant of this program. However,  $at_{-\ell_{0..4}} = 1$  is the only linear invariant produced by equations (T)+(I). ■

### Example: Producer-Consumer

Let us consider program PROD-CONS-SV (Fig. 2.23) for implementing a producer-consumer system using semaphores.

**local**  $r$ ,  $ne$ ,  $nf$ : integer where  $r = 1$ ,  $ne = N$ ,  $nf = 0$   
 $b$  : list of integer where  $b = \Lambda$

*Prod* ::  $\left[ \begin{array}{l} \text{local } x: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \text{request } ne \\ \ell_3: \text{request } r \\ \ell_4: b := b \bullet x \\ \ell_5: \text{release } r \\ \ell_6: \text{release } nf \end{array} \right] \end{array} \right]$

||

*Cons* ::  $\left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{request } nf \\ m_2: \text{request } r \\ m_3: (y, b) := (\text{hd}(b), \text{tl}(b)) \\ m_4: \text{release } r \\ m_5: \text{release } ne \\ m_6: \text{consume } y \end{array} \right] \end{array} \right]$

Fig. 2.23. Program PROD-CONS-SV (producer-consumer with shared variables).

Unlike previous versions of the Producer-Consumer program that communicated by message passing (e.g., program PROD-CONS of Fig. 1.19 (page 133)), this version communicates via the shared list variable  $b$ . To ensure that no more than  $N$  elements are ever placed in  $b$ , the program uses two semaphore variables. Variable  $ne$  counts the number of empty slots in list  $b$ . Initially,  $b$  is empty and therefore  $ne = N$ . On placing a new message into  $b$ , process *Prod* decrements  $ne$  by 1. On removing a message from  $b$ , process *Cons* increments  $ne$  by 1.

Variable  $nf$  counts the number of occupied (full) slots in  $b$ . Initially it is 0 and it is incremented by 1 whenever a new message is added to  $b$  and is decremented by 1 whenever a message is removed.

Semaphore variable  $r$  ensures that handling the shared variable is done ex-

clusively by either *Prod* or *Cons* but not by both at the same time.

We wish to prove several safety properties of this program. The relevant safety requirements are summarized by

$$\square \left( \underbrace{\neg(at\_l_4 \wedge at\_m_3)}_{\psi_1} \wedge \underbrace{at\_l_4 \rightarrow |b| < N}_{\psi_2} \wedge \underbrace{at\_m_3 \rightarrow |b| > 0}_{\psi_3} \right),$$

where

- $\psi_1: \neg(at\_l_4 \wedge at\_m_3)$

ensures mutual exclusion between the sections that manipulate the shared data structure  $b$ ;

- $\psi_2: at\_l_4 \rightarrow |b| < N$

ensures that when the producer process is about to add an element to the shared buffer  $b$ , the current size of  $b$  is strictly smaller than  $N$ , guaranteeing that the buffer never overflows;

- $\psi_3: at\_m_3 \rightarrow |b| > 0$

guarantees the symmetric requirement that the buffer never underflows, i.e., the consumer never attempts to remove an element from an empty buffer.

These three properties can be established using linear invariants derived by the methods discussed in this section.

First, we observe that the variables  $r$ ,  $ne$  (“nonempty”),  $nf$  (“non-full”), being semaphore variables with nonnegative initial values, remain invariantly non-negative. Also the value of  $|b|$ , being a length, is invariantly nonnegative. These basic facts are expressed by the invariant

$$\varphi_0: r \geq 0 \wedge ne \geq 0 \wedge nf \geq 0 \wedge |b| \geq 0.$$

The variables that are linearly incremented in this program are  $\{r, ne, nf, |b|\}$ . Note that the addition and removal of elements from the buffer  $b$  respectively increment and decrement its length  $|b|$ . This illustrates that the body of a linear invariant may contain an expression, such as  $|b|$ , in addition to program variables.

Program PROD-CONS-SV contains two cycles, to which we refer as  $C_{\text{prod}}$  and  $C_{\text{cons}}$ .

- *Bodies*

The increments of the four variables along these two cycles are as follows:

For  $C_{\text{prod}}$ :

$$\begin{aligned}\Delta(r, C_{\text{prod}}) &= 0, & \Delta(ne, C_{\text{prod}}) &= -1, & \Delta(nf, C_{\text{prod}}) &= 1, \\ \Delta(|b|, C_{\text{prod}}) &= 1,\end{aligned}$$

For  $C_{\text{cons}}$ :

$$\begin{aligned}\Delta(r, C_{\text{cons}}) &= 0, & \Delta(ne, C_{\text{cons}}) &= 1, & \Delta(nf, C_{\text{cons}}) &= -1, \\ \Delta(|b|, C_{\text{cons}}) &= -1.\end{aligned}$$

Consequently, if we contemplate linear invariants with the body

$$a_r \cdot r + a_e \cdot ne + a_f \cdot nf + a_b \cdot |b|,$$

we obtain the following cyclic equations:

$$a_r \cdot 0 - a_e + a_f + a_b = 0 \quad (\text{for } C_{\text{prod}})$$

$$a_r \cdot 0 + a_e - a_f - a_b = 0 \quad (\text{for } C_{\text{cons}}).$$

This set of equations has three linearly independent solutions. As a basis for these solutions, we may choose the following three combinations:

$$(1) \quad a_r = 1, \quad a_e = a_f = a_b = 0$$

$$(2) \quad a_e = a_f = 1, \quad a_r = a_b = 0$$

$$(3) \quad a_e = a_b = 1, \quad a_r = a_f = 0.$$

We will develop three invariants corresponding to these three solutions.

The bodies of the invariants corresponding to these solutions are

$$B_1: \quad r$$

$$B_2: \quad ne + nf$$

$$B_3: \quad ne + |b|.$$

- *Compensation expressions*

The values of the coefficients  $b_\ell$  can be computed and are presented in the table of Fig. 2.24 for the three solutions represented by  $\varphi_1$ ,  $\varphi_2$ , and  $\varphi_3$ .

- *Right constants*

By substituting the initial values of the variables, that is,

$$r = 1 \quad ne = N \quad nf = 0 \quad \text{and} \quad |b| = 0,$$

we obtain the following right-hand side constants:

$$K_1 = 1 \cdot 1 = 1$$

$$K_2 = 1 \cdot N + 1 \cdot 0 = N$$

$$K_3 = 1 \cdot N + 1 \cdot 0 = N.$$

	— Prod —			— Cons —			
	$\varphi_1$	$\varphi_2$	$\varphi_3$		$\varphi_1$	$\varphi_2$	$\varphi_3$
$b_{\ell_1}$	0	0	0	$b_{m_1}$	0	0	0
$b_{\ell_2}$	0	0	0	$b_{m_2}$	0	1	0
$b_{\ell_3}$	0	1	1	$b_{m_3}$	1	1	0
$b_{\ell_4}$	1	1	1	$b_{m_4}$	1	1	1
$b_{\ell_5}$	1	1	0	$b_{m_5}$	0	1	1
$b_{\ell_6}$	0	1	0	$b_{m_6}$	0	0	0

Fig. 2.24. Coefficients  $b_{\ell_i}$  and  $b_{m_j}$ .

- *The invariants*

Using the table of Fig. 2.24 and combining the compensation expressions into location sets, we obtain the following invariants:

$$\varphi_1: r + at_{-}\ell_{4,5} + at_{-}m_{3,4} = 1$$

$$\varphi_2: ne + nf + at_{-}\ell_{3..6} + at_{-}m_{2..5} = N$$

$$\varphi_3: ne + |b| + at_{-}\ell_{3,4} + at_{-}m_{4,5} = N.$$

There is no need to verify the invariance of these assertions, since they were developed according to our systematic construction.

We will now show that the invariants indeed imply the requirements  $\psi_1$ ,  $\psi_2$ , and  $\psi_3$ .

- *Establishing  $\psi_1$ :*

To establish  $\psi_1$ , we observe that

$$\underbrace{r + at_{-}\ell_{4,5} + at_{-}m_{3,4} = 1}_{\varphi_1} \wedge r \geq 0 \rightarrow \underbrace{\neg(at_{-}\ell_4 \wedge at_{-}m_3)}_{\psi_1}.$$

Were  $at_{-}\ell_4 \wedge at_{-}m_3$  to hold, then  $at_{-}\ell_4 = at_{-}m_3 = 1$ . By  $\varphi_1$ ,  $r = -1$ , contradicting  $r \geq 0$ .

- *Establishing  $\psi_2$ :*

To establish  $\psi_2$ , we observe that

$$\underbrace{ne + |b| + at_{-}\ell_{3,4} + at_{-}m_{4,5} = N}_{\varphi_3} \wedge ne \geq 0 \rightarrow \underbrace{at_{-}\ell_4 \rightarrow |b| < N}_{\psi_2}.$$

This is because, by  $\varphi_3$  and assuming that  $at_{-}\ell_4 = 1$ , we have

$$|b| = N - ne - 1 - at_{-}m_{4,5} \leq (N - 1) < N,$$

using the fact that  $ne \geq 0$ .

- *Establishing  $\psi_3$ :*

To establish  $\psi_3$ , we observe that

$$\underbrace{ne + nf + at_{\{3..6\}} + at_{\{2..5\}} = N}_{\varphi_2} \wedge \\ \underbrace{ne + |b| + at_{\{3,4\}} + at_{\{4,5\}} = N}_{\varphi_3} \wedge nf \geq 0 \rightarrow \underbrace{at_{\{m_3\}}}_{\psi_3} \rightarrow |b| > 0.$$

This is because, when we substitute  $at_{\{m_3\}} = 1$  in  $\varphi_2$  and  $\varphi_3$ , we obtain

$$ne + nf + at_{\{3..6\}} + 1 = N \\ ne + |b| + at_{\{3,4\}} + 0 = N.$$

From this we can obtain, by equating the left-hand sides,

$$ne + |b| + at_{\{3,4\}} = ne + nf + at_{\{3..6\}} + 1,$$

which yields

$$|b| = nf + (at_{\{3..6\}} - at_{\{3,4\}}) + 1 \geq 1,$$

using the obvious fact that  $at_{\{3,4\}} \leq at_{\{3..6\}}$ , since  $\{3,4\} \subseteq \{3..6\}$ , and that  $nf \geq 0$ . ■

In Problems 2.11–2.15, the reader is requested to derive linear invariants for several programs.

## 2.5 Completeness

Rule INV (Fig. 1.5, page 92) reduces the problem of proving the validity of a formula  $\square p$  over a fair transition system  $P$  to the task of finding an assertion  $\varphi$  that satisfies premises I1, I2, and I3.

Assume that we have spent some time in search of such an assertion  $\varphi$  but have failed to find one. There can be several reasons for such a failure.

The most obvious possibility is that  $\square p$  is not  $P$ -valid. To account for this possibility, it is recommended that the prover search in parallel for a counterexample, i.e., a computation that reaches a state in which  $p$  is false.

A second possible reason is that while  $\square p$  is  $P$ -valid, there does not exist an assertion  $\varphi$  satisfying the premises of rule INV. This possibility implies that rule INV is not universally applicable for proving all  $P$ -valid invariants, and perhaps we should be looking for additional proof rules.

In this section, we shall show that this second possibility never arises and that rule INV is, in fact, universally applicable. This is stated by the following theorem:

**Theorem 2.5** (completeness of rule INV)

For every assertion  $p$  such that  $\Box p$  is  $P$ -valid, there exists an assertion  $\varphi$  such that the premises of rule INV are provable from state validities.

Recall that premises I1–I3 require that certain implications be  $P$ -state valid. The theorem claims that these  $P$ -state validities can be proven, freely using any needed state validity as an assumption.

A general completeness proof of a given rule is expected to show that if an instance of the rule's consequence is true then the premises of the rule can be proven from a set of axioms, using a set of inference rules. The question is what is the starting point and what true facts are taken as axioms.

Since this book is focused on verifying programs by temporal reasoning, we adopt a rather advanced starting point and take all valid first-order assertions as axioms. This does not mean that we consider proofs of first-order validities a trivial task, but, rather, that we prefer to concentrate on the temporal part of the proof, assuming the necessary first-order validities as given, and showing how to use them for inferring temporal validities.

This type of limited completeness, which only covers the part of the proof beyond first-order reasoning, is called *completeness relative to first-order reasoning*. All of the completeness results in this book are of this type.

To prove this theorem, assume in the following that  $\Box p$  is  $P$ -valid. Equivalently,  $p$  holds over every  $P$ -accessible state. We will show the existence of an assertion  $\varphi$  such that the premises of rule INV are state validities.

### The Assertion $acc_P$

Consider a transition system  $P$  with system variables  $\bar{y} = (y_1, \dots, y_m)$ .

The proof of the claim is based on the construction of the *accessibility assertion*  $acc_P(\bar{y})$ , which characterizes the  $P$ -accessible states.

Assume that we have defined an assertion  $acc_P$ , such that for any state  $s$ ,

$$s \text{ is } P\text{-accessible} \quad \text{iff} \quad s \Vdash \underbrace{acc_P}_{\varphi}.$$

The details of the definition of  $acc_P$  will be described later. We will show that taking  $acc_P$  for  $\varphi$  leads to the satisfaction of the premises of rule INV. Consider each premise in turn.

For premise I1, we show that if  $s \Vdash acc_P$ , then  $s \Vdash p$ . Assume that  $s \Vdash acc_P$ . By the properties of  $acc_P$ , this implies that  $s$  is  $P$ -accessible. Since  $p$  holds over every  $P$ -accessible state, it holds in particular over  $s$ , leading to  $s \Vdash p$ . This shows that  $acc_P \rightarrow p$  is state valid.

For premise I2, let  $s$  satisfy  $s \Vdash \Theta$ . By definition,  $s$  is  $P$ -accessible and therefore must satisfy  $s \Vdash acc_P$ . This shows that  $\Theta \rightarrow acc_P$  is state valid.

For premise I3, we show that every state  $s$  satisfies  $s \Vdash \rho_\tau \wedge acc_P \rightarrow acc'_P$ , where we can present  $acc'_P$  as  $acc_P(\bar{y}')$ . Let  $s$  be a state satisfying  $s \Vdash \rho_\tau \wedge acc_P$ . By the fact that  $s \Vdash acc_P$  and the properties of  $acc_P$ ,  $s$  is  $P$ -accessible. Let  $s'$  be a  $\bar{y}$ -variant of  $s$  such that  $s'[\bar{y}] = s[\bar{y}']$ . That is,  $s'$  agrees with  $s$  on the interpretation of all variables other than  $\bar{y}$  and, for every  $i = 1, \dots, m$ ,  $s'[y_i] = s[y'_i]$ . By the properties of  $\rho_\tau$  and the fact that  $s \Vdash \rho_\tau$ , the state  $s'$  is a  $\tau$ -successor of  $s$ . Since  $s$  is  $P$ -accessible and  $s'$  is its  $\tau$ -successor, it follows that  $s'$  is also  $P$ -accessible and, therefore,  $s' \Vdash acc_P$ . As  $s[\bar{y}'] = s'[\bar{y}]$ , this implies that  $s \Vdash acc'_P$ . This shows that  $\rho_\tau \wedge acc_P \rightarrow acc'_P$  is state valid for every  $\tau \in T$ .

**Example** Consider a simple transition system EVEN with the following components:

*System variables:*  $y$  ranging over  $\mathbb{Z}$  (the integers)

*Initial condition:*  $\Theta: y = 0$

*Transitions:*  $T = \{\tau_I, \tau\}$ , where  $\rho_\tau: y' = y + 2$ .

Let  $p$  be the  $P$ -state valid assertion

$$p: y \geq 0.$$

For this simple transition system, accessibility is specifiable by

$$acc_{\text{EVEN}}(y): y \geq 0 \wedge even(y),$$

characterizing the nonnegative even integers.

It is straightforward to show that premises I1 and I2 of rule INV hold when we take  $acc_{\text{EVEN}}$  for  $\varphi$ .

Premise I3 for the diligent transition  $\tau$  assumes the form

$$\underbrace{y' = y + 2}_{\rho_\tau} \wedge \underbrace{y \geq 0 \wedge even(y)}_{acc_{\text{EVEN}}} \rightarrow \underbrace{y' \geq 0 \wedge even(y')}_{acc'_{\text{EVEN}}},$$

which is obviously state valid. ■

The preceding discussion showed that if we can construct the assertion  $acc_P$ , then rule INV is complete. It only remains to show how  $acc_P$  is defined.

## Extended Assertion Language: The Single-Variable Case

For simplicity, we will only consider the case in which all the system variables range over a single data domain  $D$ . The extension to the case of several data domains is straightforward.

As a first step in the definition of the accessibility assertion  $acc_P$ , we assume that our underlying assertion language includes the data type of *arrays*. As explained in page 170, one-dimensional arrays over  $D$  can be viewed as functions  $a: [1..n] \rightarrow D$  from a finite interval of natural numbers  $[1..n]$  into the data domain  $D$ . We refer to  $n$  as the *size* of the array.

For simplicity, we consider first the case that the transition system  $P$  has a single system variable  $y$  and every transition relation has the form  $\rho_\tau(y, y')$ . For this case, the assertion  $acc_P(y)$  is defined by

$$acc_P(y): \exists n > 0 \ \exists a \in [1..n] \mapsto D : init \wedge last \wedge evolve,$$

where the three subformulas appearing in the definition are

$$init: \Theta(a[1])$$

$$last: a[n] = y$$

$$evolve: \forall i: 1 \leq i < n: \bigvee_{\tau \in T} \rho_\tau(a[i], a[i+1]).$$

The formula  $acc_P$  states the existence of a positive integer  $n$  and an array  $a$  of size  $n$ . The intended meaning of the array  $a$  is that it records the complete history of some computation that leads to the state defined by the current value of  $y$ . If this state is  $P$ -accessible then at least one such computation must exist. Thus, array  $a$  represents a computation  $s_1, \dots, s_n$  where  $a[i]$  represents the value of  $y$  at state  $s_i$ .

- Subformula *init* states that  $a[1]$ , the first element of  $a$ , represents a value that the system variable  $y$  may assume in an initial state.
- Subformula *last* states that the last element of  $a$  is equal to the value of  $y$ .
- Subformula *evolve* states that every two consecutive elements of the array,  $a[i]$  and  $a[i + 1]$ , are related by a transition relation  $\rho_\tau$  for some transition  $\tau \in T$ .

It is not difficult to see that for any value  $d \in D$ ,  $acc_P(d) = \top$  iff  $d$  is a possible value of  $y$  in a  $P$ -accessible state.

**Example** For the transition system **EVEN** considered in the preceding example, the described construction for the assertion  $acc_P$  yields

$$(\exists n > 0) (\exists a \in [1..n] \mapsto \mathbb{Z}): \left( \begin{array}{l} a[1] = 0 \wedge a[n] = y \\ \quad \wedge \\ \forall i: 1 \leq i < n: a[i+1] = a[i] + 2 \end{array} \right)$$

In the construction of this formula, the disjunction in subformula *evolve* is taken only over the diligent transitions of system **EVEN** which consist of the single transition  $\tau$ . It is sufficient to consider only diligent transitions since a state  $s_n$  is reachable from  $s_1$  by a computation segment iff it can be reached by a computation segment that does not use the idling transition.

This formula can be simplified to

$$(\exists n > 0) (\exists a \in [1..n] \mapsto \mathbb{Z}): \left( \begin{array}{l} a[n] = y \\ \quad \wedge \\ \forall i: 1 \leq i \leq n: a[i] = 2 \cdot (i - 1) \end{array} \right),$$

which, in turn, can be simplified to

$$y \geq 0 \wedge \text{even}(y).$$

Obviously, this assertion precisely characterizes the values that  $y$  may assume in  $P$ -accessible states of transition system **EVEN**. ■

## The Multivariate Case

To deal with transition systems with several system variables, we use two-dimensional arrays. A *two-dimensional* array over a domain  $D$  is a two-argument function  $a: [1..n] \times [1..m] \mapsto D$ , mapping pairs of integers  $(i, j)$ ,  $1 \leq i \leq n$  and  $1 \leq j \leq m$ , to elements of  $D$ .

For a fixed value of the first index,  $i_0$ ,  $1 \leq i_0 \leq n$ , we can consider the one-dimensional array, consisting of the elements  $a[i_0, 1], \dots, a[i_0, m]$ . We refer to this subarray as the  $i_0$ -th *row* of  $a$ , and denote it by  $a[i_0]$ .

Let  $P$  be a transition system with system variables  $\bar{y} = y_1, \dots, y_m$ . Consider an array  $a \in [1..n] \times [1..m] \mapsto D$ . For  $i = 1, \dots, n$ , we introduce the abbreviations

$$\Theta(a[i]): \Theta(a[i, 1], \dots, a[i, m])$$

$$\rho_\tau(a[i], a[i+1]): \rho_\tau(a[i, 1], \dots, a[i, m]; a[i+1, 1], \dots, a[i+1, m]),$$

obtained by substituting  $a[i, 1], \dots, a[i, m]$  for  $y_1, \dots, y_m$  in  $\Theta$  and  $\rho_\tau$ , and substituting  $a[i+1, 1], \dots, a[i+1, m]$  for  $y'_1, \dots, y'_m$  in  $\rho_\tau$ .

We also write  $a[n] = \bar{y}$  as an abbreviation for

$$a[n, 1] = y_1 \wedge \dots \wedge a[n, m] = y_m.$$

The assertion  $\text{acc}_P(\bar{y})$  can be defined for this more general case by

$$\text{acc}_P(\bar{y}): (\exists n > 0) (\exists a \in [1..n] \times [1..m] \mapsto D): \text{init} \wedge \text{last} \wedge \text{evolve},$$

where

$$\text{init: } \Theta(a[1])$$

$$\text{last: } a[n] = \bar{y}$$

$$\text{evolve: } \forall i: 1 \leq i < n: \bigvee_{\tau \in T} \rho_\tau(a[i], a[i+1]).$$

**Example** Consider a transition system FACT defined by

*System variables:*  $y, z$ , ranging over the natural numbers  $\mathbb{N}$

*Initial condition:*  $\Theta: y = 1 \wedge z = 1$

*Transitions:*  $T = \{\tau_I, \tau\}$ , where  $\rho_\tau: y' = y + 1 \wedge z' = (y + 1) \cdot z$ .

The accessibility formula for this system is

$(\exists n > 0)(\exists a \in [1..n] \times [1, 2] \mapsto \mathbb{N}):$

$$\left( \begin{array}{l} a[1, 1] = 1 \wedge a[1, 2] = 1 \wedge a[n, 1] = y \wedge a[n, 2] = z \\ \quad \wedge \\ \forall i: 1 \leq i < n: a[i+1, 1] = a[i, 1] + 1 \wedge a[i+1, 2] = (a[i, 1] + 1) \cdot a[i, 2] \end{array} \right)$$

As before, we only considered the single diligent transition  $\tau$ .

This formula can be simplified to

$$(\exists n > 0)(\exists a \in [1..n] \times [1, 2] \mapsto \mathbb{N}): \left( \begin{array}{l} a[n, 1] = y \wedge a[n, 2] = z \\ \quad \wedge \\ \forall i: 1 \leq i \leq n: a[i, 1] = i \wedge a[i, 2] = i! \end{array} \right)$$

which, in turn, can be simplified to

$$y \geq 1 \wedge z = y!.$$

It is not difficult to see that this assertion precisely characterizes the  $P$ -accessible states for the transition system FACT. ■

## How Useful is the Assertion $\text{acc}_P$ ?

A reader who sees for the first time a completeness proof such as the one we presented may be puzzled by what seems to be a discrepancy. In Chapters 1 and 2, we repeatedly emphasized how difficult (yet important) it is to find an inductive assertion  $\varphi$  strengthening an invariant assertion  $p$ . Most of the discussion in these chapters is dedicated to the description of various techniques and heuristics for the construction of such an inductive assertion. Yet, in the last section we define a single formula  $\text{acc}_P$  which is claimed to be an inductive strengthening of *any*

invariant assertion  $p$ . Why can't we forget all the preceding techniques and always use  $\text{acc}_P$  for  $\varphi$ ?

There are two answers to this question. First, while  $\text{acc}_P$  is always inductive, the fact that it implies (strengthens)  $p$  depends on the assumption that  $p$  is *known* to be invariant. If this is known beforehand then there is no need to apply rule INV at all. If we start out without this *a priori* knowledge and our goal is to convince any possible skeptic that  $p$  is invariant, we cannot use this assumption and must establish  $\text{acc}_P \rightarrow p$  independently.

Second,  $\text{acc}_P$  is a much more complex assertion than any of those we constructed for the examples in earlier sections, which were custom-designed for the specific programs considered. The simplifications shown in examples EVEN and FACT of this section, by which the assertion  $\text{acc}_P$  was reduced to a much simpler assertion, are exceptions rather than the rule. For most programs such simplifications are not readily available.

In conclusion, while the completeness result indicates that rule INV is strong enough to be universally applicable, we should not interpret it as a recommendation to use  $\text{acc}_P$  as the assertion  $\varphi$  in rule INV.

## **The Extensions are not Essential**

For the expression of the assertion  $\text{acc}_P$  we required an assertion language that included the data type of arrays. This extension was done for purely didactic reasons but is not essential for many cases.

It is sufficient to assume that our underlying first-order language includes the data type of the natural numbers with the constants 0 and 1, and the operations of integer addition and integer multiplication. We also assume that our set of state validities includes all first-order formulas that employ the symbols  $\langle 0, 1, +, \times \rangle$  and are valid over the natural numbers.

A standard technique of recursion theory shows that every finite sequence of natural numbers  $a_1, \dots, a_n$ ,  $n > 0$ , can be encoded in a pair of natural numbers  $c$  and  $d$ , enabling the retrieval of any desired element of the sequence. With this encoding and decoding (retrieval), it is possible to express the assertion  $\text{acc}_P(\bar{y})$  without using arrays for the case of a single system variable ranging over the natural numbers.

More generally, if all the data structures used in program  $P$  can be encoded by natural numbers (e.g., the integers, the rationals, lists, or trees of integers), then it is possible to express  $\text{acc}_P(\bar{y})$  without the need for arrays, even for the multivariate case.

## 2.6 Finite-State Algorithmic Verification

For a restricted class of programs (transition systems), the question of whether any given assertion  $p$  is invariant over a program  $P$  can be determined algorithmically. This is the class of finite-state programs.

A program  $P$  is called a *finite-state program* if each system variable  $x \in V$  assumes only finitely many values in all computations of  $P$ . In some cases we can syntactically determine that a program is finite state. For example, if all data variables of the program are declared to be boolean, then the program is necessarily finite state. Another simple case is that in which all assignments have constants on their right-hand side.

```
local y1, y2: boolean where y1 = F, y2 = F
      s      : integer where s = 1
```

$$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \ell_1: \text{noncritical} \\ \quad \ell_2: (y_1, s) := (\text{T}, 1) \\ \quad \ell_3: \text{await } \neg y_2 \vee s \neq 1 \\ \quad \ell_4: \text{critical} \\ \quad \ell_5: y_1 := \text{F} \end{array} \right]$$

||

$$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad m_1: \text{noncritical} \\ \quad m_2: (y_2, s) := (\text{T}, 2) \\ \quad m_3: \text{await } \neg y_1 \vee s \neq 2 \\ \quad m_4: \text{critical} \\ \quad m_5: y_2 := \text{F} \end{array} \right]$$

Fig. 2.25. Program MUX-PET1 (Peterson's algorithm for mutual exclusion) — version 1.

**Example** (Peterson's algorithm — version 1)

Consider program MUX-PET1 presented in Fig. 2.25, which represents Peterson's algorithm for mutual exclusion without semaphores (previously presented in

Fig. 1.13, page 121).

The only assignments in this program assign 1 or 2 to variable  $s$  and  $T$  or  $F$  to the boolean variables  $y_1$  and  $y_2$ . The system variables of this program are  $\pi$ ,  $y_1$ ,  $y_2$ , and  $s$ . Since  $\pi$  must contain one location of process  $P_1$  and one from  $P_2$ , it can assume at most 36 different values. Since each of  $y_1$ ,  $y_2$ , and  $s$  can assume at most two different values, there can be at most  $36 \times 8 = 288$  different states for this program. Only 42 of these are accessible, i.e., can be reached by legal transitions from the initial state. ■

Given a finite-state program  $P$ , we can construct the *state-transition graph*  $G_P$  corresponding to  $P$ . This is a directed graph whose nodes are all the  $P$ -accessible states, and whose edges connect node  $s$  to node  $s'$  iff  $s'$  is a successor of  $s$ .

#### **Algorithm** TRANSITION-GRAFH — constructing a state-transition graph

To incrementally construct the state-transition graph  $G_P$  for a given finite-state program  $P$ , do the following:

- Initially, place as nodes in  $G_P$  all the states that are initial, i.e., states which satisfy  $\Theta$ .
- Repeat the following step until no new nodes or new edges can be added to  $G_P$ .
  - **Step:** for some  $s \in G_P$ , let  $s_1, \dots, s_k$  be the successors of  $s$ . Add to  $G_P$  all nodes among  $\{s_1, \dots, s_k\}$  that are not already there and draw a (directed) edge connecting  $s$  to  $s_i$ , for each  $i = 1, \dots, k$ .

This incremental construction can be applied to any program  $P$ , regardless of whether  $P$  is known a priori to be finite state or not. If  $P$  is finite state, then termination of the construction is guaranteed. On the other hand, if the construction terminates, this establishes the fact that  $P$  is finite state.

#### **Example** (mutual exclusion by semaphores)

Consider program MUX-SEM presented in Fig. 2.26.

This program does not satisfy the syntactic conditions stated above as sufficient for finiteness. It has an integer variable  $y$  that is modified by a **release**  $y$  statement that can, if applied repeatedly, lead to unbounded integers.

Yet, we optimistically apply the incremental construction to this program. There is only one initial state  $\langle \{\ell_0, m_0\}, 1 \rangle$ . Systematic exploration of its successors and their successors, and so on, terminates with the graph presented in Fig. 2.27. This graph represents a state  $\langle \pi: \{\ell_i, m_j\}, y: v \rangle$  by the triple  $\langle \{\ell_i, m_j\}, v \rangle$ . Due to the idling transition  $\tau_I$ , every node in the graph of Fig. 2.27

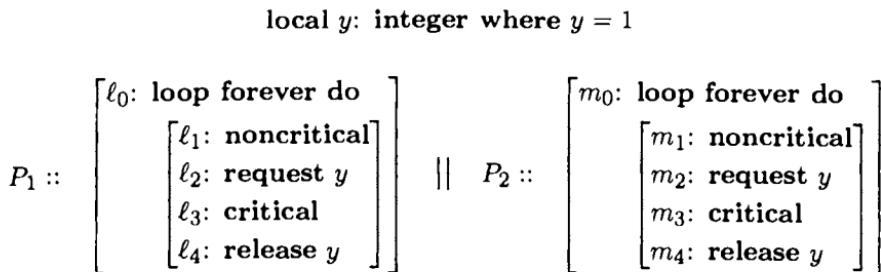


Fig. 2.26. Program MUX-SEM (mutual exclusion by semaphores).

should have been connected to itself by an edge labeled by  $\tau_i$ . However, to simplify the presentation in this figure and all subsequent representations of state transition graphs, we consistently omit these self-connecting edges. The curved arrow entering state  $\langle \{\ell_0, m_0\}, 1 \rangle$  in the graph identifies it as the initial state. ■

### Checking $P$ -Invariances

Let  $P$  be a finite-state program and  $p$  an assertion. We are interested in an algorithm that will determine whether  $p$  is  $P$ -invariant, i.e., whether  $p$  holds on all  $P$ -accessible states. The algorithm is straightforward. We start by constructing the state-transition graph  $G_P$  for  $P$ . This construction identifies the set of all accessible states as those that appear in  $G_P$ . Then, we check whether each accessible state satisfies  $p$ . If they all do, then obviously  $p$  is  $P$ -invariant.

#### Example (mutual exclusion by semaphores)

Consider again program MUX-SEM of Fig. 2.26. We may, for example, check the assertions

$$\varphi_0: y = 0 \vee y = 1$$

$$\varphi_1: \neg(at_{-\ell_3} \wedge at_{-m_3})$$

$$\varphi_2: at_{-\ell_{3,4}} + at_{-m_{3,4}} + y = 1$$

on the states appearing in the graph of Fig. 2.27. If we do so, we find that each of the three assertions holds over all the accessible states. This shows that all three are invariants of program MUX-SEM. ■

#### Example (Peterson's algorithm for mutual exclusion)

In a similar way, we present in Fig. 2.28 the state-transition graph for program

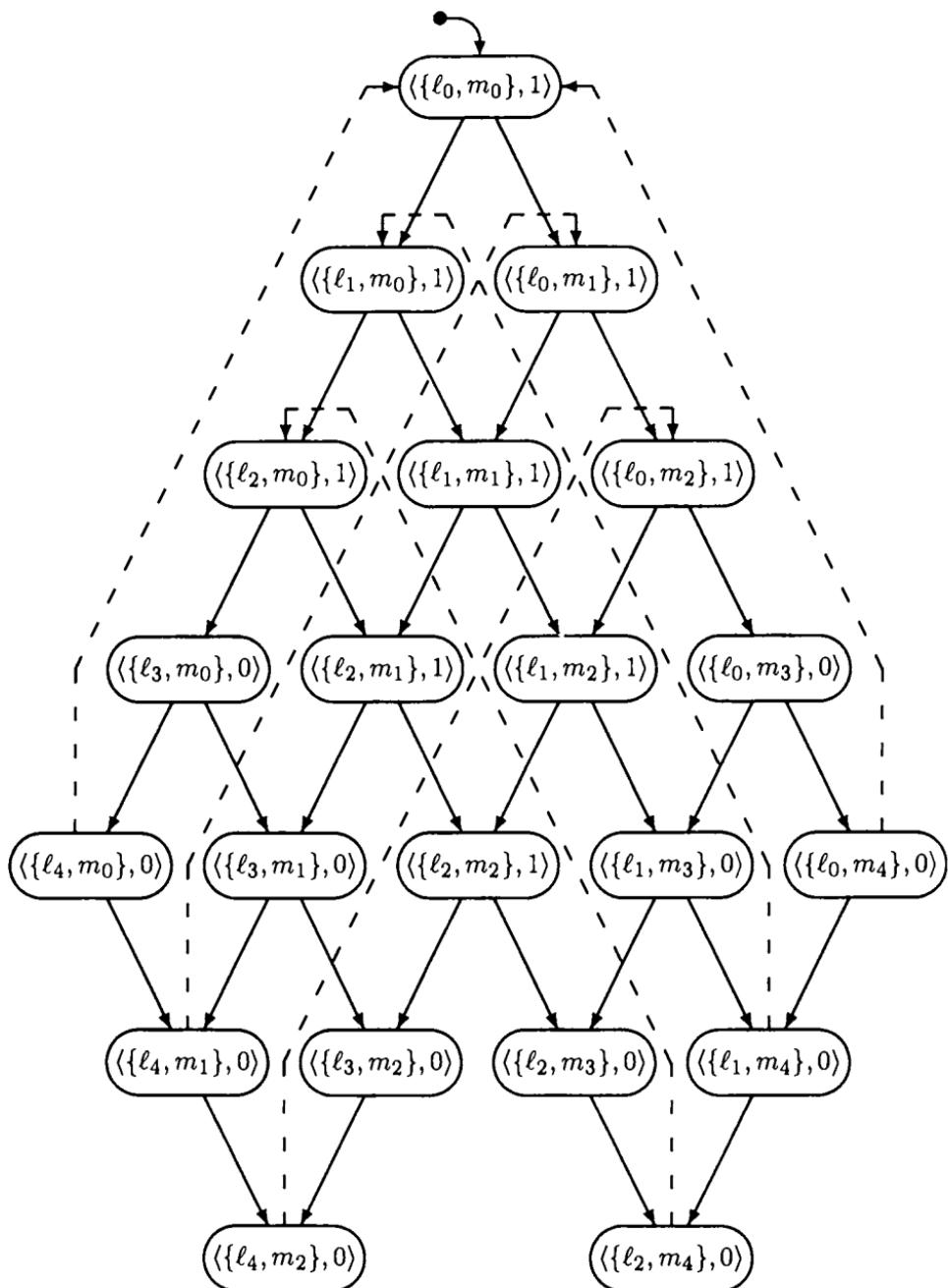


Fig. 2.27. State-transition graph for MUX-SEM.

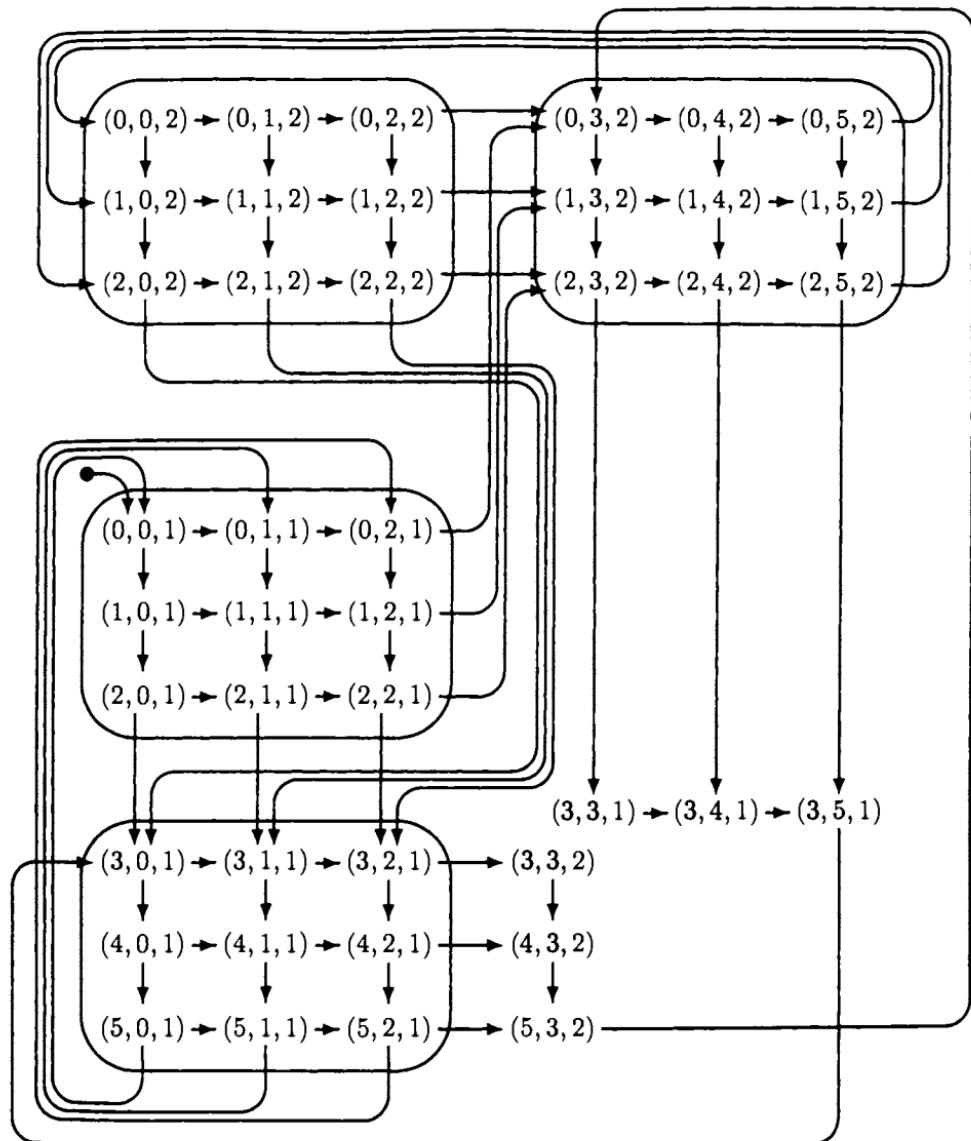


Fig. 2.28. State-transition graph for MUX-PET1.

MUX-PET1 of Fig. 2.25 (page 227). Each state in this graph lists the values  $(i, j, v)$  representing the interpretation  $\pi: \{\ell_i, m_j\}$  and  $s: v$ . The values of  $y_1$  and  $y_2$  in the states are determined by  $i$  and  $j$ , respectively. Variable  $y_1$  has the value T iff  $3 \leq i \leq 5$ . Similarly,  $y_2 = T$  iff  $3 \leq j \leq 5$ . It is straightforward to check that the following assertions hold over all accessible states:

- $$\begin{aligned}\psi_1: \quad & \neg(at\_l_4 \wedge at\_m_4) \\ \psi_2: \quad & at\_l_3 \wedge \neg at\_m_3 \rightarrow s = 1 \\ \psi_3: \quad & \neg at\_l_3 \wedge at\_m_3 \rightarrow s = 2.\end{aligned}$$

In **Problem 2.16**, the reader is requested to verify a program that performs garbage collection.

## Problems

**Problem 2.1** (asynchronous versions of RES-ND and RES-MP) page 184

Programs RES-ND and RES-MP, presented in Fig. 2.7 (page 181) and Fig. 2.8 (page 182), solved the single-resource allocation problem by synchronous message passing. Propose versions of these two programs that use asynchronous message passing. Prove that your proposed programs maintain mutual exclusion.

**Problem 2.2** (mutual exclusion with central manager) page 188

Program PMUX-MAN of Fig. 2.29 implements mutual exclusion, using a central allocator process  $A$  which receives requests in variable  $y$  and responds in variable  $x$ . Using the parameterized methods presented in Section 2.1, prove mutual exclusion for this program.

Note that this program cannot guarantee individual accessibility but only communal accessibility (see page 31).

\* **Problem 2.3** (Peterson's algorithm for  $n$  processes) page 188

Program MUX-PET-N of Fig. 2.30 presents a solution to the mutual-exclusion problem for an arbitrary number of processes.

The *while* statement at  $\ell_3$  gradually increases the priority of process  $P[i]$  from 1 to  $n$ . When the priority, expressed by  $j$  ( $j[i]$  for process  $P[i]$ ), grows beyond  $n$ ,  $P[i]$  is admitted to its critical section. On entering priority level  $j$ , process  $P[i]$  records (at  $\ell_4$ ) its current priority in  $y[i]$  and writes its identity number ( $i$ ) in  $s[j]$  the signature logbook for this level. It then remains at level  $j$  until it detects one of two occurrences. Either the number of processes of priority not smaller than  $j$  becomes 1, which means that  $P[i]$  is the only one with such priority, or the logbook  $s[j]$  assumes a value different than  $i$ , implying that some of the processes entered level  $j$  after  $P[i]$ .

To check for the first occurrence, statements  $\ell_8$  to  $\ell_{13}$  repeatedly count the number of processes with priority not smaller than  $j$ . Note that one cannot assume that on termination of the *while* statement  $\ell_{10}$  the value of *above-me* necessarily equals the number of processes with priority not smaller than  $j$ . While  $P[i]$  counts,

**local**  $x, y$ : integer **where**  $x = 0, y = 0$

$$A :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{while } y = 0 \text{ do} \\ \quad \left[ \begin{array}{l} \ell_2: \text{skip} \\ \ell_3: x := y \\ \ell_4: \text{while } x \neq 0 \text{ do} \\ \quad \left[ \begin{array}{l} \ell_5: \text{skip} \\ \ell_6: y := 0 \end{array} \right] \end{array} \right] \end{array} \right]$$

||

$$\left| \begin{array}{c} M \\ \parallel \\ j=1 \end{array} \right. P[j] :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \text{while } x \neq j \text{ do} \\ \quad \left[ \begin{array}{l} m_3: y := j \\ m_4: \text{critical} \\ m_5: x := 0 \end{array} \right] \end{array} \right] \end{array} \right]$$

Fig. 2.29. Program PMUX-MAN.

some processes with high priority may go through the critical section and lower their priority, and some processes with low priority may raise their priority to the tested level after being considered.

Prove that program MUX-PET-N guarantees mutual exclusion to each of its processes, by showing that the assertion  $N_{15} \leq 1$  is an invariant of the program.

**Problem 2.4** (ungrouping communication statements) page 191

Many programs contain processes with the following structure:

$$P_k :: \left[ \begin{array}{l} \text{while } \varphi \text{ do} \\ \quad \left[ \begin{array}{l} \text{OR}_{i=1}^n [\langle \text{when } p_i \text{ do } com_i \rangle; S_i] \\ \text{or} \\ \text{OR}_{j=1}^m T_j \end{array} \right] \end{array} \right]$$

where  $com_i$  is a *send* or *receive* statement, statements  $S_1, \dots, S_n$  and  $T_1, \dots, T_m$

in  $n$  : integer where  $n \geq 2$   
 local  $y, s$ : array[1..n] of integer where  $y = s = 0$

$\prod_{i=1}^n P[i] ::$

```

local j, k, above-me: integer
l0: loop forever do
  l1: noncritical
  l2: j := 1
  l3: while j < n do
    l4: y[i] := j
    l5: s[j] := i
    l6: above-me := n
    l7: while above-me > 1  $\wedge$  s[j] = i do
      l8: above-me := 0
      l9: k := 1
      l10: while k  $\leq$  n do
        l11: if y[k]  $\geq$  j then
          l12: above-me := above-me + 1
        l13: k := k + 1
      l14: j := j + 1
    l15: critical
    l16: y[i] := 0
  
```

Fig. 2.30. Program MUX-PET-N (mutual exclusion for  $n$  processes).

do not contain any communication substatements (*send* or *receive*), and all variable references are to variables that are local to the process. An example of a process with this structure is process  $S[i]$  of program SERV-RING (Fig. 2.32) below. The grouped *when* statement ensures that the communication  $com_i$  takes place only when the condition  $p_i$  holds.

- Show that the grouped *when* statements are not essential. That is, construct a statement congruent to  $P_k$  that does not use grouped statements. A statement is considered to be congruent to  $P_k$  if it may replace  $P_k$  in any context of a larger program and yield the same final results on termination (for precise definitions, see Problem 0.3).

- Apply this transformation to process  $S[i]$  of program SERV-RING (Fig. 2.32).

**\*\* Problem 2.5** (resource allocation by token ring) page 191

In Fig. 2.31 we present an interconnection scheme for program SERV-RING, which is presented in Fig. 2.32.

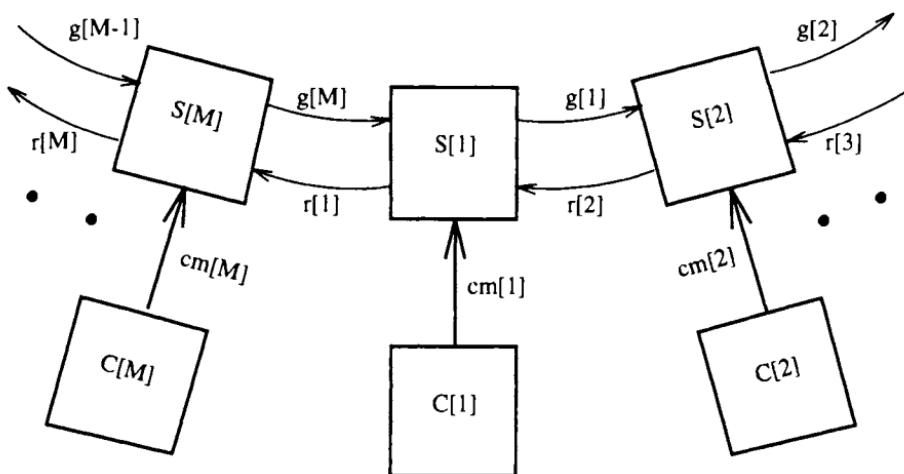


Fig. 2.31. Interconnection scheme for program SERV-RING.

The program contains a ring of server processes  $S[1], \dots, S[M]$  that transmit a token around the ring. Each server process  $S[i]$ ,  $i = 1, \dots, M$ , is connected to a customer process  $C[i]$ , which may need the resource for its activity. Process  $C[i]$  communicates with its server process  $S[i]$  via synchronous channel  $cm[i]$ . As we see,  $C[i]$  sends the value 1 to  $S[i]$  to signal a request for the resource. The resource is granted when the next communication statement  $\ell_3$  is executed. Release of the resource is signalled by statement  $\ell_5$ .

The ring of channels  $g[1], \dots, g[M]$  is used to move the token in a counterclockwise direction. The ring of channels  $r[1], \dots, r[M]$  is used to transmit requests for the token in a clockwise direction. Process  $S[i]$  uses the following local variables:

- has** — A boolean variable which is true in  $S[i]$  whenever  $S[i]$  holds the token. It is to be expected that  $\sum_{i=1}^M has[i] = 1$ , implying that precisely one process  $S[i]$  has the token at any state.
- t\_reqd** — A boolean variable which is true in  $S[i]$  if  $S[i \oplus_M 1]$  has recently requested the token from  $S[i]$  via a message on channel  $r[i]$ .
- status** — A variable that ranges over  $[0..2]$ . This variable represents the status of services currently requested and provided by server  $S[i]$  to customer  $C[i]$ . A value of 0 implies that  $C[i]$  has no interest in the resource. A value of 1 represents a state in which  $C[i]$  has requested the resource.

**in**  $M$  : integer where  $M > 1$   
**r, g:** array [1.. $M$ ] of channel of boolean  
**cm:** array [1.. $M$ ] of channel of [0..2]

$$\text{SERV-RING} :: \left[ \bigparallel_{i=1}^M S[i] \quad \bigparallel \quad \bigparallel_{i=1}^M C[i] \right]$$

where the program for  $S[i]$ ,  $i = 1, \dots, M$ , is given by:

**local**  $has$  : boolean where  $has = (i = 1)$   
 $t\_reqd$ : boolean where  $t\_reqd = F$   
 $status$  : [0..2] where  $status = 0$

$m_0$ : **loop forever do**

$[m_1^a: \langle \text{when } \neg t\_reqd \text{ do } r[i] \Rightarrow t\_reqd \rangle$   
**or**  
 $m_1^b: \langle \text{when } \neg has \wedge (status = 1 \vee t\_reqd) \text{ do } r[i \ominus_M 1] \Leftarrow T \rangle$   
**or**  
 $[m_1^c: g[i \ominus_M 1] \Rightarrow has; m_2: \text{skip}]$   
**or**  
 $m_1^d: \left\langle \begin{array}{l} \text{when } has \wedge status = 0 \wedge t\_reqd \text{ do } g[i] \Leftarrow T; \\ (has, t\_reqd) := (F, F) \end{array} \right\rangle$   
**or**  
 $m_1^e: \langle \text{when } has \vee status \neq 1 \text{ do } cm[i] \Rightarrow status \rangle$

and the program for  $C[i]$ ,  $i = 1, \dots, M$ , is given by:

$C[i] :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \ell_1: \text{noncritical} \\ \ell_2: cm[i] \Leftarrow 1 \\ \ell_3: cm[i] \Leftarrow 2 \\ \ell_4: \text{critical} \\ \ell_5: cm[i] \Leftarrow 0 \end{array} \right]$

Fig. 2.32. Program SERV-RING (resource allocation by token ring).

and is waiting for it. A value of 2 represents a state in which  $C[i]$  has acquired the resource and is currently using it.

- \* (a) The principal safety property of program SERV-RING is that at most one customer process may be at its critical section. This can be expressed by the assertion

$$\varphi_1: \sum_{i=1}^M at\_l_4[i] \leq 1.$$

Prove that  $\varphi_1$  is an invariant of SERV-RING.

- \*\* (b) An advantage of program SERV-RING over other resource-allocation programs is that the token moves around only when there is a need for it. An assertion that captures part of this property is

$$\varphi_2: at\_m_2[i] \rightarrow \bigvee_{j=1}^M at\_l_3[j].$$

This assertion states that if we observe server  $S[i]$  at location  $m_2$ , which means that it has just received the token from  $S[i \ominus M 1]$ , then some customer process must be waiting for the resource. Prove that  $\varphi_2$  is an invariant of program SERV-RING.

#### \* Problem 2.6 (detecting distributed termination) page 201

Consider a system of processes arranged along a ring. The interconnection scheme of the processes is presented in Fig. 2.33. Program RING-TERM of Fig. 2.34 presents an algorithm for detecting termination in such a system.

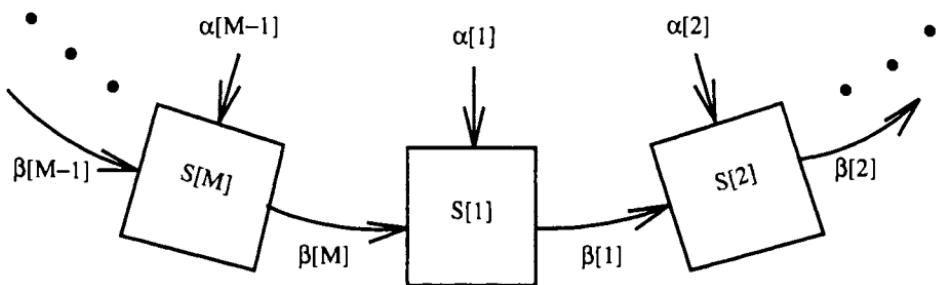


Fig. 2.33. Interconnection scheme of program RING-TERM.

Processes in RING-TERM communicate by two arrays of synchronous channels. Array  $\alpha[1..M]$  represents transmission of data values between processes. Each process  $P[i]$ ,  $i = 1, \dots, M$ , has an incoming channel  $\alpha[i]$  which is used by all other processes  $P[j]$ ,  $j \neq i$ , to send data values to  $P[i]$ . Array  $\beta[1..M]$  forms a

in  $M$ : integer where  $M > 0$   
 $\alpha$  : array [1.. $M$ ] of channel of integer  
 $\beta$  : array [1.. $M$ ] of channel of [0..2]

local term, active, changed: boolean where  
 $term = F, active = changed = T$   
has : boolean where has = ( $i = 1$ )  
token : [0..2] where token = 0  
 $x, y$  : integer

$\ell_0$ : while  $\neg term$  do

$\ell_1^a$ : when active do  $[\ell_2$ : produce  $x$ ;  $\ell_3$ :  $\bigvee_{j \neq i} \alpha[j] \Leftarrow x]$

or

$\ell_1^b$ :  $\alpha[i] \Rightarrow y$ ;  $\ell_4$ : (active, changed) := (T, T);  
 $\ell_5$ : consume  $y]$

or

$\ell_1^c$ : active := F

or

$\ell_1^d$ :  $\beta[i \ominus_M 1] \Rightarrow token$ ;  $\ell_6$ : has := T

or

$\ell_1^e$ :  $\left\langle \begin{array}{l} \text{await } has \wedge \neg active \wedge i \neq 1 \\ \text{if } changed \text{ then } token := 0 \\ (changed, has) := (F, F) \\ \beta[i] \Leftarrow token \\ \text{if } token = 2 \text{ then } term := T \end{array} \right\rangle$

or

$\ell_1^f$ :  $\left\langle \begin{array}{l} \text{await } has \wedge \neg active \wedge i = 1 \wedge token < 2 \\ \beta[1] \Leftarrow token + 1 \\ has := F \end{array} \right\rangle$

or

$\ell_1^g$ :  $\left\langle \begin{array}{l} \text{await } has \wedge \neg active \wedge i = 1 \wedge token = 2 \\ (term, has) := (T, F) \end{array} \right\rangle$

$\ell_7$ :

Fig. 2.34. Program RING-TERM (detecting termination along a ring).

ring of channels where, for each  $i = 1, \dots, M$ ,  $\beta[i]$  connects process  $P[i]$  to process  $P[i \oplus_M 1]$ .

We assume that processes  $P[1], \dots, P[M]$  are engaged in some application that has the following characteristics:

- Each process runs its own activity which may involve sending and receiving messages via the  $\alpha$ -channels.
- At some point, process  $P[i]$  may realize it has no further tasks to perform and move to an inactive state. In program RING-TERM, this is expressed by setting local variable *active* to F.
- An inactive process does not initiate any outputs to other processes. However, it is still ready to accept inputs from other processes, as shown in statement  $\ell_1^b$ . Receiving an input from another process reactivates the receiving process and may require it to engage in additional activity, possibly outputting additional messages to other processes. Consequently, on receiving an input, statement  $\ell_4$  sets *active* to T, implying that the process is once more active.

Obviously, processes may switch in and out of the active state in arbitrary patterns. However, if at any point all  $M$  processes are inactive, the system has terminated. This is because, in such a state, no process can initiate an output. We refer to this situation as a state of *global termination*. The purpose of the algorithm represented by program RING-TERM is to detect a situation of global termination, if it ever occurs, and cause the complete program to terminate in such a situation.

In order to detect global termination, the program uses the  $\beta$ -channels and several auxiliary variables. The algorithm operates by passing a token around the ring. The token may assume an integer value in the range [0..2]. A token value of 2 signifies that a global termination has been achieved. The local boolean variable *has* is true in a process if the process currently has the token. In that case, variable *token* represents its value. Initially,  $has[1] = T$  and  $has[i] = F$  for all  $i \neq 1$ , implying that process  $P[1]$  has the token.

Consider the following proposal for detecting global termination: starting with  $P[1]$ , each process that has the token and is inactive ( $active = F$ ) sends the token to its right neighbor. When  $P[1]$  receives the token from  $P[M]$ , it knows that each  $P[i]$ ,  $i = 1, \dots, M$ , has been inactive at least once.

Unfortunately, this does not guarantee global termination. This is because, after becoming inactive and passing the token on, a process may receive a message and awake again. The algorithm of program RING-TERM uses a more stringent criterion. Process  $P[1]$  realizes that global termination has been achieved when it has evidence that the token has gone around the ring twice and each process  $P[i]$ ,  $i = 2, \dots, M$  has been continuously inactive between the two last visits of the token to  $P[i]$ .

To gather this evidence, the program uses the local variable *changed*. This variable is reset to F within grouped statement  $\ell_1^e$ , just before process  $P[i]$  sends the token on  $\beta[i]$ . Then, whenever  $P[i]$  receives a data message, variable *changed* is set to T at  $\ell_4$ . Thus, the value of *changed* for  $P[i]$  is intended to be F iff  $P[i]$  has been continuously inactive since it last sent the token on  $\beta[i]$ . Process  $P[1]$  always sends a token with a positive value within grouped statement  $\ell_1^f$ . Any other process resets the value of the token to 0 within  $\ell_1^e$  if *changed* = T.

Thus, if  $P[1]$  receives a token with value 1 via  $\beta[M]$ , it means that the variable *changed* = F for all  $P[i]$ ,  $i = 2, \dots, M$ . In that case,  $P[1]$  sends the token again with value 2. After receiving and sending a token with value 2, each process  $P[i]$ ,  $i = 2, \dots, M$ , terminates by setting variable *term* to T at  $\ell_1^e$ . When  $P[1]$  receives a token of 2, it also terminates.

It is obvious that program TERM-RING can deadlock. For example, a system with  $M = 2$  can deadlock if both processes reach  $\ell_3$  at the same time. In such a state, each of the processes attempts to send a message to the other process, but no process is ready to receive.

However, we should view algorithm TERM-RING as consisting of a termination-detection layer, represented by statements  $\ell_1^c - \ell_1^g$ , superimposed on an application layer, represented by statements  $\ell_1^a$  and  $\ell_1^b$ . The previously described deadlock is entirely within the application layer. It can be shown that the termination-detection layer does not introduce additional deadlocks into the system.

The principal safety property of this program is that, if any of the processes believes that global termination has been achieved, then this is indeed the case. This is stated by the following assertion

$$\varphi: \bigvee_{i=1}^M \text{term}[i] \rightarrow \bigwedge_{i=1}^M \neg \text{active}[i].$$

Show that assertion  $\varphi$  is an invariant of program RING-TERM.

#### \* Problem 2.7 (asynchronous termination detection) page 201

Program RING-TERM of Fig. 2.34 detects global termination of a ring of processes that communicate by synchronous message passing. Consider the same problem for the case that channels  $\alpha[1..M]$  and  $\beta[1..M]$  are asynchronous.

Program RING-TERM does not provide a satisfactory solution for this case. It is possible that all processes have decided on local termination (set their *active* variable to F) but some of the  $\alpha$ -channels still contain pending messages. Such a situation should not be considered as global termination since the process to which these messages are directed may still read them and reactivate itself.

One way to handle the asynchronous case is for each process to maintain an integer variable  $C$ , which we refer to as the *message count*. At any state of the

execution,  $C[i]$  (variable  $C$  for process  $P[i]$ ) should hold the number of messages sent by  $P[i]$  on any  $\alpha[j]$ ,  $j \neq i$ , minus the number of messages received by  $P[i]$  via  $\alpha[i]$ . Obviously, a state of global termination is one in which all processes are inactive and  $\sum_{i=1}^M C[i] = 0$ . A zero sum of the message counts implies that no  $\alpha$ -channel is currently containing any pending messages.

To check whether the sum of message counts is zero, it is proposed that messages transferred on the  $\beta$ -channels be records of the form

$\langle ftoken: [0..2], sum: integer \rangle$

The field  $ftoken$  has the same meaning as  $token$  in program RING-TERM. The field  $sum$  contains the sum of message counts in the processes preceding the receiving process. Thus, process  $P[i]$  receives a message with a value of  $sum$  which is an approximation of  $\sum_{j=1}^{i-1} C[j]$ . It is an approximation, since after contributing to  $sum$ , process  $P[j]$ , for some  $j < i$ , may send or receive additional messages, changing the value of  $C[j]$ . If  $P[i]$ ,  $i > 1$ , receives a certain value of  $sum$  it will send in the grouped statement corresponding to  $\ell_1^e$  the record  $\langle token, sum + C[i] \rangle$  on channel  $\beta[i]$ . Process  $P[1]$  should send within statement  $\ell_1^f$  the record  $\langle token + 1, C[1] \rangle$  on  $\beta[1]$ .

(a) Present a version of program RING-TERM for detecting global termination of asynchronously communicating processes, based on the preceding suggestions.

(b) Prove that assertion

$$\varphi: \bigvee_{i=1}^M term[i] \rightarrow \bigwedge_{i=1}^M (\neg active[i] \wedge |\alpha[i]| = 0)$$

is an invariant of your program.

### Problem 2.8 (parallel fixpoint computation) page 201

Consider program PAR-FIX of Fig. 2.35, which computes a fixpoint of the set of equations

$$y_1 = f_1(\bar{y}), \dots, y_M = f_M(\bar{y}),$$

where  $\bar{y} = (y_1, \dots, y_M)$ . The computation is done in parallel.

The boolean array  $conv[1], \dots, conv[M]$  is used to record which equation  $y[i] = f_i(\bar{y})$  has been checked to be true. Initially, all variables  $conv[1], \dots, conv[M]$  are false. Process  $P[i]$  checks at  $\ell_1$  whether the equation  $y[i] = f_i(\bar{y})$  holds. If the equation holds, then  $conv[i]$  is set to  $T$ . Otherwise,  $y[i]$  is assigned the new value  $f_i(\bar{y})$  and  $conv[1], \dots, conv[M]$  are set to  $F$  in parallel. The iterations of each  $P[i]$  are continued as long as there exists some false  $conv[j]$ .

(a) Prove for this program the invariance of the following assertion:

in  $M$  : integer where  $M > 0$   
 local  $conv$ : array [1.. $M$ ] of boolean where  $conv = F$   
 out  $y$  : array [1.. $M$ ] of integer

$\ell_0$ : while  $\bigvee_{j=1}^M \neg conv[j]$  do  
 $\quad \left[ \begin{array}{l} \ell_1^a: \langle \text{when } y[i] = f_i(\bar{y}) \text{ do } conv[i] := T \rangle \\ \text{or} \\ \ell_1^b: \langle \text{when } y[i] \neq f_i(\bar{y}) \text{ do } y[i] := f_i(\bar{y}) \rangle \\ \ell_2: \bigvee_{j=1}^M Q[i, j] :: \left[ \begin{array}{l} \ell_3: conv[j] := F \\ \ell_4: \end{array} \right] \end{array} \right]$   
 $\ell_5:$

Fig. 2.35. Program PAR-FIX (parallel computation of a fixpoint).

$$\varphi: \bigwedge_{i=1}^M conv[i] \rightarrow \bigwedge_{i=1}^M y[i] = f_i(\bar{y}).$$

(b) Consider a version of program PAR-FIX in which grouped statement  $\ell_1^a$  is replaced by the statement

$[\ell_1^a: \text{when } y[i] = f_i(\bar{y}) \text{ do } \ell_6: conv[i] := T]$

and statement  $\ell_1^b$  is replaced by the statement

$[\ell_1^b: \text{when } y[i] \neq f_i(\bar{y}) \text{ do } \ell_7: y[i] := f_i(\bar{y})]$ .

Is assertion  $\varphi$  also an invariant of the modified program? Prove or show a counterexample.

**Problem 2.9** (cyclic equations are sufficient) page 214

Let  $P$  be a cyclic program. Show that any solution of the cyclic equations (CE), specifying values for the unknowns  $a_1, \dots, a_r, b_{\ell_i^j}, K$ , satisfies the sets of equations (T)+(I).

**Problem 2.10** ((T)+(I) implies (CE)+ $\bar{\chi}$ ) page 214

Let  $P$  be a cyclic program. Prove that any solution of equations (T)+(I) is a

linear combination of a solution of (CE) and the assertions  $\chi_1, \dots, \chi_m$  presented on page 214.

**Problem 2.11** ( $N$  semaphores) page 220

Consider program SEM-N presented in Fig. 2.36, which generalizes program SEM-3 of Fig. 2.21.

```

in      M: integer where M > 1
local y : array [1..M] of integer where y[1] = 1,
                                         y[2] = ... = y[M] = 0

          ||| P[i] :: [l0: loop forever do
          i=1           [l1: noncritical
                         l2: request y[i]
                         l3: critical
                         l4: release y[i ⊕_M 1]]]

```

Fig. 2.36. Program SEM-N (mutual exclusion with  $M$  semaphores).

Apply the method for constructing linear invariants to derive an invariant that implies the property of mutual exclusion. This property can be expressed by the invariance of the following assertion:

$$\varphi_{ij}: \text{at\_}l_3[i] + \text{at\_}l_3[j] \leq 1$$

claimed for every  $i, j \in [1..M]$ ,  $i \neq j$ .

**Problem 2.12** (different invariants for PROD-CONS-SV) page 220

In the analysis of program PROD-CONS-SV (Fig. 2.23, page 216), we chose a set of independent solutions which led to the following invariant bodies

$$B_1: r, \quad B_2: ne + nf, \quad B_3: ne + |b|.$$

Consider instead a different set of solutions which replaces  $B_3$  by

$$\hat{B}_3: |b| - nf.$$

- (a) Complete  $\hat{B}_3$  into a full invariant  $\hat{\varphi}_3$ .
- (b) Show that requirements  $\psi_2$  and  $\psi_3$  can be inferred from  $\varphi_1$ ,  $\varphi_2$ , and  $\hat{\varphi}_3$ .
- (c) Show that invariant  $\varphi_3$  can be inferred from  $\varphi_1$ ,  $\varphi_2$ , and  $\hat{\varphi}_3$ .

**Problem 2.13** (variant A of Dekker's algorithm) page 220

Consider the program MUX-DEK-A in Fig. 1.29.

- (a) List the simple cycles of the program.
- (b) As written, program MUX-DEK-A has no linear variables. Rewrite MUX-DEK-A so that  $y_1$ ,  $y_2$ , and  $t$  are linear variables. Thus, for example, each assignment to  $y_1$  must be of the form  $y_1 := y_1 + C$  for some constant  $C$ .
- (c) For each variable  $v \in \{y_1, y_2, t\}$ , compute  $\Delta(v, \tau)$  for each  $\tau$  in MUX-DEK-A.

**Problem 2.14** (cycles and linear variables in MUX-DEK-A) page 220

Using the techniques of Section 2.4, prove  $\neg(at_{-\ell_7} \wedge at_{-m_7})$  for program MUX-DEK-A of Fig. 1.29 (page 156).

**Problem 2.15** (cycles and linear variables in MUX-BAK-A) page 220

Consider the program MUX-BAK-A of Fig. 1.33 (page 159).

- (a) List the simple cycles of the program.
- (b) Which variables (if any) are linear variables?

**\*\* Problem 2.16** (garbage-collection algorithm) page 232

In Fig. 2.37 we present program GARB-COLL which proposes an algorithm for garbage collection that runs in parallel with an application. Process *Mutator* in program GARB-COLL represents the application while the parallel process *Collector* represents the garbage collector. A situation that requires garbage collection can often be described as an array of records, each containing a data field(s) and several pointer fields that may point to other records in the array. The schematic representation in program GARB-COLL assumes  $N > 1$  records, which contain a single data field and two pointer fields  $D[1]$  and  $D[2]$ . Since we represent the records as an array, pointers are represented as subscripts in this array, ranging over  $[1..N]$ . The only way the application can refer to records is through the pointer variables  $r_1, \dots, r_m$  to which we refer as *roots* and also as *pointer registers*. The garbage collection algorithm of program GARB-COLL requires an additional field  $C$  in each record. We refer to this field as the *color* of the record and it assumes one of the values *B* (black) or *W* (white).

In Fig. 2.38 we present the program for the mutator. According to this schematic representation, the mutator alternates between a noncritical activity at  $\ell_1$  and an activity that may modify parts of the data structure consisting of the array  $M$  and roots  $r_1, \dots, r_m$ . The noncritical activity may freely read and write the value field (field *val*) of records pointed to by the roots. That is, it may perform operations of the form

$$M[r_i].val := f(M[r_j].val),$$

```

in    N: integer where  $N > 1$ 
local M: array [1..N] of record
      ⟨val: data, D: array [1..2] of [1..N], C: [B, W]⟩
      where  $M[1..N].C = W$ 
r1, ..., rm: [1..N]

```

*Mutator* || *Collector*

Fig. 2.37. Program GARB-COLL (garbage collection on the fly).

for arbitrary  $i, j \in \{1, \dots, m\}$ .

However, the noncritical activity cannot modify any of the other fields of any of the records, or assign values to the roots. The critical activity may engage in one of the following operations:

```

 $\ell_0$ : loop forever do
   $\ell_1$ : noncritical
    [
       $\ell_2^a$ :  $r_i := M[r_j].D[k]$ 
      or
       $\ell_2^b$ :  $M[r_j].D[k] := r_i$ 
      or
       $\ell_2^c$ :  $r_i := r_j$ 
    ]
    OR
    [
       $i, j \in [1..m]$ 
       $k \in [1, 2]$ 
       $\ell_3$ :  $M[r_i].C := B$ 
    ]
  ]

```

Fig. 2.38. Program for process *Mutator*.

- Statement  $\ell_2^a$  copies a pointer from a  $D$ -field of a record, pointed to by  $r_j$ , to pointer register  $r_i$ .
- Statement  $\ell_2^b$  copies the value of pointer register  $r_i$  to field  $D[k]$  of the record pointed to by register  $r_j$ .
- Statement  $\ell_2^c$  copies  $r_j$  to  $r_i$ .

Each of these operations is followed by execution of statement  $\ell_3$ , which blackens (colors black) the record whose pointer has been moved around.

The state of the data structure  $M + \{r_1, \dots, r_m\}$  can be viewed as a big graph  $G_M$  which has a node for each root  $r_i$ ,  $i = 1, \dots, m$ , and a node for each record  $M[p]$ ,  $p \in [1..N]$ . We draw a directed edge from  $r_i$  to  $M[p]$  if  $r_i = p$ , and a directed edge from  $M[p]$  to  $M[q]$  if  $M[p].D[k] = q$  for some  $k \in \{1, 2\}$ . The record  $M[p]$  is said to be *accessible* if there exists a path in the graph connecting  $r_i$  to  $M[p]$  for some  $i \in [1..m]$ . Since all modifying operations of the mutator can refer to records only through the pointer registers  $r_1, \dots, r_m$ , the mutator can reach only records that are accessible according to the above definition. Note also that the mutator can cause an accessible record to become inaccessible but not vice versa. Consequently, all inaccessible records may be considered as garbage and it is the role of the collector to reclaim these records and make them accessible again.

In Fig. 2.39, we present the program for the collector. The collector alternates between a marking and collecting phase. The intended meaning of the color scheme is that records whose color is black may be accessible, while records that are white at a certain stage of the marking phase are known to be inaccessible (garbage).

Initially, all records are white, and some records are colored black by the mutator at statement  $\ell_3$ . However, the collector does not fully rely on these assumptions, and statements  $m_1-m_{12}$  attempt to systematically color all accessible records black.

Statements  $m_1$  and  $m_2$  blacken all records pointed to by the root registers. Then, statements  $m_5-m_8$  sweep once over all the records and blacken all descendants of each black record. This simple-minded (and inefficient) approach may require several sweeps to propagate blackness along a chain of pointers. Assume, for example, that record  $M[4]$ , which is pointed to by one of the roots, points to  $M[3]$ , which points to  $M[2]$ , which points to  $M[1]$ . After performing  $m_1$  and  $m_2$ ,  $M[4]$  is black, but  $M[1]$ ,  $M[2]$ , and  $M[3]$  are still white. Complete execution of the *for* statement  $m_5$ , will blacken  $M[3]$  but leave  $M[1]$  and  $M[2]$  white. Two more sweeps are necessary to propagate blackness all the way to  $M[1]$ .

Consequently, the color propagating statement  $m_5$  is enclosed within the *while* statement  $m_4$  which repeatedly executes  $m_5$  until no further propagation is possible. Statements  $m_9-m_{12}$  count the number of black records. The *while* statement  $m_4$  terminates when the number of black records counted in two consecutive sweeps is the same.

The collector then moves to the collection phase. Statements  $m_{13}-m_{16}$  examine each record in turn. White records are identified as inaccessible (garbage) and are added to the free list. We assume that the free list is pointed to by one of the pointer registers, so adding a record to this list makes the record accessible. Black records are identified as accessible and are colored white, in preparation for the next marking phase.

The main safety property of this algorithm is that process *Collector* only

```

local  $i, count, Pcount: integer$ 
       $k : [1..N]$ 
       $j : [1..2]$ 

 $m_0:$  loop forever do
  [ -- Marking phase
     $m_1:$  for  $i := 1$  to  $m$  do      -- Blacken Roots
       $m_2: M[r_i].C := B$ 
     $m_3: (count, Pcount) := (0, -1)$ 
     $m_4:$  while  $count > Pcount$  do
      [ -- Propagate coloring
         $m_5:$  for  $k := 1$  to  $N$  do
          [  $m_6:$  if  $M[k].C = B$  then
            [  $m_7:$  for  $j := 1$  to 2 do
               $m_8: M[M[k].D[j]].C := B$  ]
          ]
        ]
      ]
    ]
  ]
  -- Count number of black records
   $m_9: Pcount := count$ 
   $m_{10}: count := 0$ 
   $m_{11}: \text{for } k := 1 \text{ to } N \text{ do}$ 
    [  $m_{12}: \text{if } M[k].C = B \text{ then}$ 
       $m_{13}: count := count + 1$  ]
  ]
  -- Collection phase
   $m_{14}: \text{for } k := 1 \text{ to } N \text{ do}$ 
    [  $m_{15}: \text{if } M[k].C = W \text{ then}$ 
      [  $m_{16}: \text{Append } M[k] \text{ to free list}$ 
        else  $m_{17}: M[k].C := W$  ]
    ]
  ]

```

Fig. 2.39. Program for process *Collector*.

collects inaccessible records. This can be stated by the invariance of the assertion

$$\varphi: at\_m_{15} \rightarrow \neg accessible(k),$$

where  $accessible(k)$  is an assertion which holds for  $k \in [1..M]$  iff there exists a

path in the graph  $G_M$ , connecting some root  $r_i$  to  $M[k]$ .

Show that  $\varphi$  is an invariant of program GARB-COLL.

## Bibliographic Remarks

**Parameterized programs:** Our approach to the presentation and uniform verification (i.e., a single proof for any number of processes  $M > 0$ ) of parameterized programs follows Marina and Pnueli [1991b]. However, the notation for a parallel composition of a family of processes, whose size depends on a parameter  $M$ , has been widely used in most texts dealing with concurrent programs, e.g., Milner [1980], Hoare [1984], Chandy and Misra [1988], Apt and Olderog [1991], and Francez [1992].

Program MAX-ARRAY of Fig. 2.5 (page 173) is taken from Cormen, Leiserson, and Rivest [1990]. Complexity analyses of algorithms for the parallel computation of the maximal element in an array are presented in Valiant [1975] and Cook, Dwork, and Reischuk [1986].

The single-resource allocation problem, which is equivalent to the mutual-exclusion problem, was introduced in Dijkstra [1968a]. He proposed the use of semaphores as synchronization constructs for solving this problem. The centralized solution presented in programs RES-MP and RES-SV follows more abstract programs presented in Andrews [1991]. The readers-writers problem was introduced in Courtois, Heymans and Parnas [1971], where it was solved using semaphores. Our solution in program READ-WRITE uses generalized semaphores that are based on the *Pchunk* and *Vchunk* operations discussed in Andrews [1991]. The multiple resource allocation was first introduced through its canonical example — the *dining philosophers* problem — in Dijkstra [1968a]. The two solutions presented in programs DINE-CONTR and DINE-EXCL, which ensure starvation freedom, are also discussed in Ben-Ari [1990] and Andrews [1991]. An extension of the problem, called the *drinking philosophers problem*, considers a more general situation in which processes require an arbitrary number of resources that may be shared between more than two processes. This problem was posed and elegantly solved in Chandy and Misra [1984].

**Deadlock prevention:** The observation that requesting resources in an order compatible with some total order is a necessary condition for deadlock prevention (and also sufficient if we use semaphores) was made by Havender [1968] who devised the resource-ordering scheme for the IBM OS/360 system, and also considered by Collier [1968]. A more general problem in which several units of the same resource are available (quantifiable resources) and processes request a certain number of units, has been considered in Dijkstra [1968a] for the case of a single type of resource. This has been generalized to several quantifiable resources

in Habermann [1969]. Both algorithms were included as part of the "THE" operating system, as described in Dijkstra [1968b].

For additional discussions on deadlocks and their prevention, we refer readers to Holt et al. [1978] and Peterson and Silberschatz [1985].

**Constructing linear invariants:** Our approach to the calculation of linear invariants is based on Francez [1976] and its elaboration in Manna and Pnueli [1981b]. A more comprehensive analysis of linear invariants, which also considers linear inequalities, is presented in Cousot and Halbwachs [1978]. A similar analysis based on forming a set of linear equations is proposed in Karr [1976] and generalized by Granger [1991]. Very similar algorithms are used in Petri nets to compute  $S$ -invariants, which were introduced in Lautenbach [1972]. The use of  $S$ -invariants for Petri nets and their construction are described in Reisig [1985]. Additional details of the construction of these invariants are described in Martinez and Silva [1982]. Clarke [1979] showed that the method of linear invariants is incomplete in the sense that not all properties of (even finite-state) programs can be established by linear invariants.

**Completeness:** The notion of relative completeness used in the proof of completeness of rule INV is due to Cook [1978] who proves relative completeness of a proof system for sequential programs. The use of the assertion  $acc_P$ , characterizing the accessible states, is inspired by a similar construction also appearing in Cook's paper. Cook's notion of relative completeness is usually stated generically for a family of possible interpretations, requiring each of them to be *expressive*, which, for sequential programs, essentially means the ability to express the transformation of a state by a loop. A closely related concept is that of *arithmetical completeness* presented in Harel [1979], where the interpretation is fixed to contain at least Peano Arithmetic. Since this text assumes a fixed assertion language that includes the "real" integers, the difference between the two concepts is less significant. Our construction of  $acc_P$  is certainly based on the integers, so it is fair to say that the completeness theorem in Section 2.5 establishes the arithmetical completeness of rule INV. However, it is straightforward to extend it to a true relative completeness in the sense of Cook, provided an interpretation is considered *expressive* if it can express the assertion  $acc_P$ . Proofs of completeness, relative to the denotational semantics of sequential programs, are presented in de Bakker [1980], whose Appendix B (written by J.I. Zucker) shows that any interpretation containing first-order Peano Arithmetic is expressive. The interested reader may consult Apt [1981], Apt and Olderog [1991], and Francez [1992] for a more detailed discussion and comparison of these two notions of completeness.

Relative completeness for a proof system for assertional verification of concurrent programs was established in Owicki and Gries [1976a], Apt, Francez, and de Roever [1980], and Levin and Gries [1981]. Previous proofs of relative completeness for temporal logic were given in Manna and Pnueli [1983a, 1991a].

The technique of encoding finite sequences and arrays of natural numbers as a pair of natural numbers, using integer addition and multiplication, is based on Gödel's  $\beta$ -predicate using the Chinese Remainder Theorem. Details can be found in Enderton [1972], pages 248–249.

**Algorithmic verification:** Algorithms for checking whether a propositional temporal formula is valid over a finite-state system were proposed at about the same time in Clarke and Emerson [1981] and in Queille and Sifakis [1982]. Both were based on a branching-time temporal logic. More references to general model-checking algorithms will be given in the bibliographic discussion of Chapter 5. The very simple algorithm given in this chapter amounts to reachability analysis, in which we construct all the accessible states of a finite-state system. Such simple algorithms are considered in many contexts, including the analysis of Petri nets and other representations of finite-state systems such as Kyng [1983], Diaz [1982], Queille and Sifakis [1982], and presented in texts such as Reisig [1985] and Peterson [1981]. A finite-state verification system which contains many memory-efficient techniques for generating and exploring the state-transition graph is reported by Holzmann [1991]. Another tool for the verification of state invariant properties over finite-state systems is Mur $\phi$ , described in Dill et al. [1992].

**Algorithms presented in the Problems Section:** The distributed termination problem for a ring of asynchronously communicating processes is considered and solved in Dijkstra, Feijen, and van Gasteren [1983]. This is the basis for Problem 2.7 and its synchronous version, Problem 2.6. The algorithm for a complete communication graph first appeared in Misra [1983], which also explains how to modify the algorithm for an arbitrary communication graph. Other approaches to the termination detection problem are presented in Francez [1980], Dijkstra and Scholten [1980], Misra and Chandy [1982], Francez, Rodeh, and Sintzoff [1981], and Mattern [1987].

The main ideas of the garbage-collection algorithm of Problem 2.16 were proposed in Dijkstra et al. [1978]. The actual algorithm presented is taken from Ben-Ari [1984], which only uses two colors in comparison to the four colors used in Dijkstra et al. [1978].

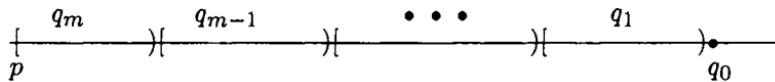
# Chapter 3

## Precedence

In this chapter we present methods for verifying safety properties that are expressed by *nested waiting-for* (nested unless) formulas of the general form

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0,$$

for state formulas (assertions)  $p, q_0, q_1, \dots, q_m$ . Recall (see page 48) that a computation satisfies this formula if every  $p$ -position starts a succession of intervals, that can be summarized by the following figure:



Each  $q_i$ -interval,  $1 \leq i \leq m$ , is a set of successive positions which all satisfy  $q_i$ , and may be empty or extend to infinity.

As will be shown in Section 4.8, every nested waiting-for formula is a safety formula, i.e., it is equivalent to a formula of the form  $\square \psi$ , for some past formula  $\psi$ .

Waiting-for formulas are very useful for expressing a rich class of *precedence* properties, such as absence of unsolicited response, order preservation of messages, and bounded overtaking.

### 3.1 Waiting-for Rule

#### Basic Waiting-for Rule

We first consider methods for proving the  $P$ -validity of *simple waiting-for* (unless) formulas, which have the general form

$$p \Rightarrow q \mathcal{W} r,$$

where  $p$ ,  $q$ , and  $r$  are assertions. This formula states that any  $p$ -state is followed by a  $q$ -interval that may be terminated only by an occurrence of  $r$ . The  $q$ -interval may be empty, if the  $p$ -state also satisfies  $r$ . The  $q$ -interval may also extend to infinity, in which case no terminating  $r$ -state is needed.

Some simple waiting-for formulas can be established by the basic rule WAIT-B (Fig. 3.1).

For assertions  $\varphi$ ,  $\psi$ ,

$$\frac{\{\varphi\} \text{ } T \text{ } \{\varphi \vee \psi\}}{\varphi \Rightarrow \varphi \text{ } W \text{ } \psi}$$

Fig. 3.1. Rule WAIT-B (basic waiting-for).

The premise of rule WAIT-B requires that the assertion

$$\rho_T \wedge \varphi \rightarrow \varphi' \vee \psi'$$

be  $P$ -state valid, for every transition  $\tau \in T$  of the program  $P$ .

The full-form presentation of rule WAIT-B, explicitly displaying the type of validity assumed on each line is given by

$$\frac{P \Vdash \rho_T \wedge \varphi \rightarrow \varphi' \vee \psi' \quad \text{for each } \tau \in T}{P \models \varphi \Rightarrow \varphi \text{ } W \text{ } \psi}$$

**Justification** To justify rule WAIT-B, assume that  $\varphi$  and  $\psi$  satisfy the premise of the rule. Consider a computation  $s_0, s_1, \dots$  and assume that  $s_i$  satisfies  $\varphi$ .

By the premise,  $s_{i+1}$  must satisfy  $\varphi \vee \psi$ . If  $s_{i+1}$  satisfies  $\psi$ , the computation obviously satisfies  $\varphi \text{ } W \text{ } \psi$  at position  $i$ . Otherwise,  $s_{i+1}$  satisfies  $\varphi$  and we can repeat the argument for  $s_{i+2}$ . It follows that there exists an interval starting at  $s_i$ , such that  $\varphi$  holds throughout the interval, and the interval either extends to infinity or terminates at a state satisfying  $\psi$ .

Consequently, the computation satisfies  $\varphi \text{ } W \text{ } \psi$  at position  $i$ . ■

**Example** Consider the nondeterministic program of Fig. 3.2.

We wish to show the validity of the formula

$$\underbrace{y > 0}_{\varphi} \Rightarrow \underbrace{y > 0}_{\varphi} \text{ } W \text{ } \underbrace{y = 0}_{\psi},$$

stating that whenever  $y$  is positive, it either stays positive forever or stays positive until it becomes 0. In particular, this property excludes a change of  $y$  from a positive value to a negative value without going through zero.

**out**  $y$ : integer **where**  $y = 1$

**$\ell_0$ : loop forever do**  $\left[ \begin{array}{l} \ell_1^a: y := y + 5 \\ \text{or} \\ \ell_1^b: y := y - 1 \\ \text{or} \\ \ell_1^c: (\text{when } y < 0 \text{ do } y := y - 2) \end{array} \right]$

Fig. 3.2. A nondeterministic program.

Using rule WAIT-B, we have to establish the premise

$$\{y > 0\} \mathcal{T} \{y > 0 \vee y = 0\},$$

which is equivalent to

$$\rho_\tau \wedge y > 0 \rightarrow y' \geq 0 \quad \text{for every } \tau \text{ in } \mathcal{T}.$$

The relevant transitions are

$$\ell_1^a: \text{yields } y' = y + 5 \text{ which implies } y > 0 \rightarrow y' \geq 0.$$

$$\ell_1^b: \text{yields } y' = y - 1 \text{ which implies } y > 0 \rightarrow y' \geq 0.$$

$$\ell_1^c: \text{yields } y < 0 \wedge y' = y - 2 \text{ which implies } y > 0 \rightarrow y' \geq 0.$$

The desired conclusion follows. ■

## General Waiting-for Rule

As with the basic rule INV-B for proving invariants, the applicability of rule WAIT-B for proving an arbitrary formula of the form

$$p \Rightarrow q \mathcal{W} r$$

is limited. First, rule WAIT-B can only prove conclusions of the above form in which  $p = q$ . But more importantly, it is rarely the case that the verification condition  $\{p\} \mathcal{T} \{p \vee r\}$  is state valid.

The resolution of this difficulty for the simple waiting-for formulas is similar to its resolution by rule INV for the case of invariants. To prove  $p \Rightarrow q \mathcal{W} r$ , we

attempt to strengthen  $q$  into an assertion  $\varphi$  for which  $\{\varphi\} T \{\varphi \vee r\}$  is state valid.

This strategy is supported by the general waiting-for rule WAIT (Fig. 3.3).

For assertions  $p, q, r, \varphi$ ,

$$\begin{array}{l} W1. \quad p \rightarrow \varphi \vee r \\ W2. \quad \varphi \rightarrow q \\ W3. \quad \{\varphi\} T \{\varphi \vee r\} \\ \hline p \Rightarrow q \mathcal{W} r \end{array}$$

Fig. 3.3. Rule WAIT (general waiting-for).

**Justification** Rule WAIT is justified as follows. Assume that assertions  $p, q, r$ , and  $\varphi$  satisfy premises W1–W3 of the rule. Consider a computation with state  $s_i$ , at which  $p$  holds. Due to W1,  $s_i$  also satisfies  $\varphi \vee r$ . If it satisfies  $r$ , then obviously  $q \mathcal{W} r$  holds.

Let us assume  $s_i$  satisfies  $\varphi$ . Due to W3 and rule WAIT-B, from the point that  $\varphi$  holds, it continues holding until an occurrence of an  $r$ -state or throughout the rest of the computation. Due to W2, wherever  $\varphi$  holds, so does  $q$ . It follows that from the point  $p$  holds,  $q$  will hold continuously until an occurrence of an  $r$ -state or throughout the rest of the computation. ■

We refer to assertions  $\varphi$  and  $q$  as *intermediate assertions*, since they bridge the gap between  $p$  and  $r$ . That is, these assertions hold on all the states lying between a  $p$ -state and the next  $r$ -state, if it exists, or all the way to infinity.

Note that premise W1 can also be written as  $p \wedge \neg r \rightarrow \varphi$ . Thus, we may consider  $\varphi$  as an assertion that lies between  $q$  and  $p \wedge \neg r$ , in the sense that it is stronger than  $q$  and weaker than  $p \wedge \neg r$ .

### Example (Peterson's algorithm — version 1)

Consider program MUX-PET1 in Fig. 3.4, which presents Peterson's algorithm for mutual exclusion. We considered this program in Section 1.4 (Fig. 1.13, page 121), where we proved that it maintains mutual exclusion, expressed by the invariance of

$$\psi_1: \neg(at\_l_4 \wedge at\_m_4).$$

The proof also established the following invariants:

```
local y1, y2: boolean where y1 = F, y2 = F
      s      : integer where s = 1
```

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: (y_1, s) := (\text{T}, 1) \\ \ell_3: \text{await } \neg y_2 \vee s \neq 1 \\ \ell_4: \text{critical} \\ \ell_5: y_1 := \text{F} \end{array} \right] \end{array} \right]$

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: (y_2, s) := (\text{T}, 2) \\ m_3: \text{await } \neg y_1 \vee s \neq 2 \\ m_4: \text{critical} \\ m_5: y_2 := \text{F} \end{array} \right] \end{array} \right]$

Fig. 3.4. Program MUX-PET1 (Peterson's algorithm for mutual exclusion) — version 1.

- $\varphi_0: s = 1 \vee s = 2$
- $\varphi_1: y_1 \leftrightarrow \text{at-}\ell_{3..5}$
- $\varphi_2: y_2 \leftrightarrow \text{at-}m_{3..5}$
- $\varphi_3: \text{at-}\ell_3 \wedge \text{at-}m_4 \rightarrow y_2 \wedge s = 1$
- $\varphi_4: \text{at-}\ell_4 \wedge \text{at-}m_3 \rightarrow y_1 \wedge s = 2.$

- *Simple precedence*

Our current interest in this program is to show some precedence properties. We wish to show that when one process, say  $P_1$ , is waiting to enter the critical section, its competitor  $P_2$  may enter its critical section at most once ahead of  $P_1$ . To show this for program MUX-PET1, it is sufficient to prove the formula:

$$\psi_2: \underbrace{\text{at-}\ell_3 \wedge \text{at-}m_{0..2}}_p \Rightarrow \underbrace{\neg \text{at-}m_4}_q \mathcal{W} \underbrace{\text{at-}\ell_4}_r.$$

This formula states that if  $P_1$  has preceded  $P_2$  in getting to the waiting position

( $\ell_3$  and  $m_3$ , respectively), it will also precede  $P_2$  in entering the critical section.

Trying to match  $\psi_2$  against the conclusion of the basic waiting-for rule WAIT-B, we see that it is inapplicable. This is because  $p \neq q$  in  $\psi_2$ . Therefore, we are left with the general rule WAIT, and have to find a  $\varphi$  that lies between  $q$  and  $p \wedge \neg r$ .

- *Constructing  $\varphi$*

As a first attempt, we take  $\varphi$  to be  $p$  (equivalently  $p \wedge \neg r$ ) itself, i.e.,  $at\_\ell_3 \wedge at\_m_{0..2}$ . However, trying to show premise W3, i.e., that  $\varphi$  is preserved by all transitions except those that establish  $r$ :  $at\_\ell_4$ , we fail. The transition causing the failure is  $m_2$ , which is always enabled, and neither preserves  $p$  nor achieves  $at\_\ell_4$ .

Therefore, we must weaken  $\varphi$  to allow a situation in which  $at\_m_3$  is also possible. On the other hand, if we believe  $\psi_2$  to hold, the situation allowing  $at\_m_3$  must be restricted in a way that will disable the execution of  $m_3$ . Since  $at\_\ell_3$  already implies  $y_1 = T$ , it remains only to guarantee that  $s = 2$ . We are encouraged in this attempt to weaken  $\varphi$  by the fact that the statement  $m_2$  indeed sets  $s$  to 2. Thus, our proposal for  $\varphi$  is

$$\varphi: at\_\ell_3 \wedge (at\_m_{0..2} \vee (at\_m_3 \wedge s = 2)).$$

- *Establishing W1–W3*

We have to establish three premises:

- *Premise W1*

$$\underbrace{at\_\ell_3 \wedge at\_m_{0..2}}_p \rightarrow \underbrace{at\_\ell_3 \wedge (at\_m_{0..2} \vee \dots)}_{\varphi} \vee \underbrace{\dots}_r.$$

Clearly holds.

- *Premise W2*

$$\underbrace{\dots \wedge (at\_m_{0..2} \vee (at\_m_3 \wedge \dots))}_{\varphi} \rightarrow \underbrace{\neg at\_m_4}_q.$$

Trivially,  $\varphi$  implies  $at\_m_{0..3}$ , which implies  $\neg at\_m_4$ .

- *Premise W3*

We have to show that

$$\begin{aligned} p_T \wedge \underbrace{at\_\ell_3 \wedge (at\_m_{0..2} \vee (at\_m_3 \wedge s = 2))}_{\varphi} \rightarrow \\ \underbrace{at'\_\ell_3 \wedge (at'\_m_{0..2} \vee (at'\_m_3 \wedge s' = 2))}_{\varphi'} \vee \underbrace{at'\_\ell_4}_{r'} \end{aligned}$$

for every  $\tau \in \mathcal{T}$ . It is not difficult to identify the critical transitions for this verification condition as  $\ell_3$ ,  $m_2$ , and  $m_3$ .

- Transition  $\ell_3$   
Establishes  $at'_-\ell_4$  which is the required  $r$ .
- Transition  $m_2$   
Assertion  $\varphi$  implies  $at_-\ell_3$ , which by  $\rho_{m_2}$  ( $m_2$  being parallel to  $\ell_3$ ) leads to  $at'_-\ell_3$ . The relation  $\rho_{m_2}$  implies  $at'_-m_3 \wedge s' = 2$ . Thus,  $\rho_{m_2} \wedge \varphi$  implies  $\varphi'$ .
- Transition  $m_3$   
We show that  $\rho_{m_3} \wedge \varphi$  is false, which means that  $m_3$  is disabled under  $\varphi$ . Assertion  $\varphi$  implies  $at_-\ell_3$ , which by the invariant  $\varphi_1$  leads to  $y_1 = T$ . The relation  $\rho_{m_3}$  implies  $at_-m_3$ , which by  $\varphi$  leads to  $s = 2$ . Thus  $\rho_{m_3} \wedge \varphi$  implies  $y_1 = T$  and  $s = 2$  which violates the clause  $\neg y_1 \vee s \neq 2$  implied by  $\rho_{m_3}$ .

By rule WAIT, this proves the validity of the precedence formula  $\psi_2$ . ■

In **Problem 3.1**, we request the reader to consider a reflexive version of rules WAIT-B and WAIT.

## Systematic Derivation of Intermediate Assertions

In the preceding example, we had to identify an intermediate assertion  $\varphi$  that together with  $p$ ,  $q$ , and  $r$ , satisfied the premises of rule WAIT. We intend to show that in many cases, the process of constructing  $\varphi$  can be done in a systematic way resembling the strengthening of an assertion into an inductive one.

It is often the case that we can identify one or more transitions that when taken, lead to an  $r$ -state. We refer to these transitions as *escape* transitions since, by the requirement of premise W3 of rule WAIT, the only way we can escape from the set of states satisfying  $\varphi$  is to move to an  $r$ -state. For the example of program MUX-PET1, where  $r$  is  $at_-\ell_4$ , the only escape transition is  $\ell_3$  which establishes  $at_-\ell_4$  when taken.

Two main heuristics can be used for constructing good intermediate assertions. We refer to them as *forward propagation* and *backward propagation*, and will consider each of them separately.

### Forward Propagation

In forward propagation we try to characterize all the states that can be reached from  $(p \wedge \neg r)$ -states without taking any of the escape transitions. In many of

the considered cases  $p$  implies  $\neg r$ , so we may refer to  $(p \wedge \neg r)$ -states simply as  $p$ -states.

### Example (Peterson's algorithm — version 1)

Consider program MUX-PET1 of Fig. 3.4, where  $p$  is  $at_{-}\ell_3 \wedge at_{-}m_{0..2}$  and  $r$  is  $at_{-}\ell_4$ . Some enabled transitions, such as  $m_0$  and  $m_1$ , lead from one  $p$ -state to another. Transition  $\ell_3$  is an escape transition and should not be considered. This leaves transition  $m_2$  which leads from a  $p$ -state to a state satisfying the assertion  $p_1$ :  $at_{-}\ell_3 \wedge at_{-}m_3 \wedge s = 2$ . Let us now consider what transitions may lead from states satisfying  $p \vee p_1$ , i.e.,

$$\varphi: at_{-}\ell_3 \wedge (at_{-}m_{0..2} \vee (at_{-}m_3 \wedge s = 2)),$$

to states that do not satisfy  $p \vee p_1$ .

The only candidate nonescape transition is  $m_3$  which, by using the invariant  $\varphi_1$ :  $y_1 \leftrightarrow at_{-}\ell_{3..5}$ , can be shown to be disabled on  $(p \vee p_1)$ -states. It follows that  $\varphi$  is an adequate characterization of the states that are reachable from  $p$ -states without using any escape transitions. ■

Let  $\Phi$  be any assertion, and  $\tau$  a transition with transition relation  $\rho_\tau(V, V')$ . We define the *forward propagation* (also called the *postcondition*) of  $\Phi$  via  $\tau$  to be the formula  $\Psi$ , also denoted  $post(\tau, \Phi)$ , given by<sup>4</sup>

$$\Psi(V) = post(\tau, \Phi): \exists V^0: \Phi(V^0) \wedge \rho_\tau(V^0, V).$$

This formula characterizes all the states that are  $\tau$ -successors of a  $\Phi$ -state.

As a simple example, consider the case that  $V = \{x, y\}$ ,  $\rho_\tau$  is given by  $x' = x + y \wedge y' = x$ , and  $\Phi$  is  $x = y$ . Then  $post(\tau, \Phi)$  is given by

$$\exists x^0, y^0: \underbrace{x^0 = y^0}_{\Phi(V^0)} \wedge \underbrace{x = x^0 + y^0 \wedge y = x^0}_{\rho_\tau(V^0, V)},$$

which can be simplified to

$$\Psi: x = y + y.$$

We summarize the forward propagation method for *waiting-for* properties as follows:

### Procedure (forward propagation)

- Let  $\Phi_0$  be  $p$  (or  $p \wedge \neg r$ ).
- For  $k = 0, 1, \dots$ , repeat the following step:

<sup>4</sup> The postcondition  $post(\tau, \Phi)$  should not be confused with  $post(S)$ , the post-location of statement  $S$ , introduced in Chapter 0.

- Let  $\tau$  be some nonescape transition such that  $post(\tau, \Phi_k)$  does not imply  $\Phi_k$ .

Choose assertion  $\Psi_{k+1}$  to be  $post(\tau, \Phi_k)$  or an assertion that is implied by it.

Let  $\Phi_{k+1}$  be  $\Phi_k \vee \Psi_{k+1}$ .

This step is to be repeated until we obtain an assertion  $\Phi_t$ , such that

$$post(\tau, \Phi_t) \rightarrow \Phi_t$$

for every nonescape transition  $\tau$ .

If we have already accumulated some invariants for program  $P$  whose conjunction is  $\chi$ , we may replace any implication  $\varphi \rightarrow \psi$  appearing in the propagation procedure by the weaker implication  $\chi \wedge \varphi \rightarrow \psi$ .

There is no guarantee that the procedure terminates. However, if it does terminate, it yields a  $\Phi_t$  that, when substituted for  $\varphi$ , satisfies premises W1 and W3 of rule WAIT.

### Example (Peterson's algorithm — version 1)

Let us illustrate this process with program MUX-PET1 of Fig. 3.4. We start with

$$\Phi_0: \underbrace{at\_l_3 \wedge at\_m_{0..2}}_p .$$

For the next step we pick  $\tau: m_2$ . The formula  $post(m_2, \Phi_0)$  is given by

$$\begin{aligned} post(m_2, \Phi_0) = \exists \underbrace{(\pi^0, y_1^0, y_2^0, s^0)}_{V^0}: & \underbrace{(at\_l_3)^0 \wedge (at\_m_{0..2})^0}_{\Phi_0(V^0)} \wedge \\ & \underbrace{(at\_m_2)^0 \wedge at\_m_3 \wedge ((at\_l_3)^0 \leftrightarrow at\_l_3)}_{\rho_{m_2}(V^0, V)} \wedge s = 2 \wedge \dots . \end{aligned}$$

In this formula,  $(at\_l)^0$  stands for  $l \in \pi^0$ .

Obviously, this formula is equivalent to

$$\Psi_1: at\_l_3 \wedge at\_m_3 \wedge s = 2 .$$

To be more precise, we should have said that  $post(m_2, \Phi_0)$  is  $P$ -equivalent to  $\Psi_1$ , taking into account the various invariants known about program MUX-PET1. In particular, the equivalence relies on the invariant  $\varphi_1: y_1 \leftrightarrow at\_l_{3..5}$ .

We consequently take

$$\Phi_1: \underbrace{at\_l_3 \wedge at\_m_{0..2}}_{\Phi_0} \vee \underbrace{at\_l_3 \wedge at\_m_3 \wedge s = 2}_{\Psi_1} .$$

The process converges at this stage, since  $\Phi_1$  is preserved under all transitions except for the escape transition  $\ell_3$ .

Note that assertion  $\Phi_1$ , obtained by the application of the algorithm, is equivalent to the intermediate assertion

$$\varphi: \text{at-}\ell_3 \wedge (\text{at-}m_{0..2} \vee (\text{at-}m_3 \wedge s = 2))$$

discussed previously. ■

## Backward Propagation

A dual approach to forward propagation is *backward propagation*. While forward propagation attempts to weaken  $p$  until it becomes an assertion preserved under all nonescape transitions, backward propagation attempts to strengthen  $q$  until we reach a similar stage.

We recall that the basic step of backward propagation, namely, supplementing an assertion  $\Gamma$  with a precondition  $\text{pre}(\tau, \Gamma)$ , has already been introduced and extensively used in Section 1.4, where it was referred to as “assertion propagation.” For convenience, we repeat the definitions.

For an assertion  $\Gamma$  and a transition  $\tau$  with transition relation  $\rho_\tau(V, V')$ , we define the *backward propagation* (also called the *precondition*) of  $\Gamma$  via  $\tau$  to be the formula  $\Delta$ , also denoted  $\text{pre}(\tau, \Gamma)$ , given by

$$\Delta(V) = \text{pre}(\tau, \Gamma): \forall V': \rho_\tau(V, V') \rightarrow \Gamma(V').$$

This formula characterizes all the states that are  $\tau$ -predecessors of a  $\Gamma$ -state.

As a simple example, consider the case in which  $V = \{x, y\}$ ,  $\rho_\tau$  is given by  $x' = x + y \wedge y' = x$ , and  $\Gamma$  is  $x = y + y$ . Then  $\text{pre}(\tau, \Gamma)$  is given by

$$\forall x', y': \underbrace{x' = x + y \wedge y' = x}_{\rho_\tau(V, V')} \rightarrow \underbrace{x' = y' + y'}_{\Gamma(V')}$$

which can be simplified to  $x + y = x + x$  or equivalently

$$\Delta: y = x.$$

For the standard case where the transition relation has the form

$$\rho_\tau: C_\tau \wedge \bar{y}' = \bar{e},$$

the precondition can be presented in the simpler form

$$\Delta: C_\tau \rightarrow \Gamma[\bar{y} \mapsto \bar{e}],$$

where  $\Gamma[\bar{y} \mapsto \bar{e}]$  is obtained by replacing each occurrence of  $y_i$  in  $\Gamma$  by the corresponding  $e_i$ .

For example, if  $\rho_\tau: T \wedge x' = x + y \wedge y' = x$  and  $\Gamma$  is  $x = y + y$ , then  $\text{pre}(\tau, \Gamma)$  is given by

$$\tau \rightarrow (x = y + y)[(x, y) \mapsto (x + y, x)],$$

which can be simplified to  $x + y = x + x$  or equivalently

$$\Delta: \quad x = y.$$

We summarize the backward propagation method for *waiting-for* properties as follows:

**Procedure** (backward propagation)

- Let  $\Gamma_0$  be  $q$ .
- For  $k = 0, 1, \dots$ , repeat the following step:
  - Let  $\tau$  be some nonescape transition such that  $\Gamma_k$  does not imply  $\text{pre}(\tau, \Gamma_k)$ . Choose assertion  $\Delta_{k+1}$  to be  $\text{pre}(\tau, \Gamma_k)$  or an assertion that implies  $\text{pre}(\tau, \Gamma_k)$ . Let  $\Gamma_{k+1}$  be  $\Gamma_k \wedge \Delta_{k+1}$ .

This step is to be repeated until we obtain an assertion  $\Gamma_f$ , such that

$$\Gamma_f \rightarrow \text{pre}(\tau, \Gamma_f)$$

for every nonescape transition  $\tau$ .

If  $\chi$  is a previously established  $P$ -invariant, we may use  $\chi \wedge \varphi \rightarrow \psi$  instead of the stronger implication  $\varphi \rightarrow \psi$  throughout the propagation procedure.

There is no guarantee that the procedure terminates. However, if it does terminate, it yields an assertion  $\Gamma_f$  that, when substituted for  $\varphi$ , satisfies premises W2 and W3 of rule WAIT.

**Example** (Peterson's algorithm — version 1)

Let us illustrate this process on the case of program MUX-PET1 of Fig. 3.4. We start with

$$\Gamma_0: \quad \underbrace{\neg at\_m_4}_{q}.$$

For the next step we pick  $\tau$ :  $m_3$ . The formula  $\text{pre}(m_3, \Gamma_0)$  is given by

$$\forall V': \quad \underbrace{at\_m_3 \wedge (\neg y_1 \vee s \neq 2) \wedge at'_m_4 \wedge \dots}_{\rho_{m_3}(V, V')} \rightarrow \underbrace{\neg at'_m_4}_{\Gamma_0(V')}$$

where the omitted part of  $\rho_{m_3}$  contains only conjuncts of the form  $y'_1 = y_1 \wedge s' = s \wedge \dots$

Since  $\rho_{m_3}$  implies  $at'_m_4$ , this formula expresses the requirement that  $m_3$  is disabled. Thus, we obtain

$$\Delta_1: \text{at\_}m_3 \rightarrow (y_1 \wedge s = 2).$$

Taking

$$\Gamma_1: \underbrace{\neg \text{at\_}m_4}_{\Gamma_0} \wedge \underbrace{\text{at\_}m_3 \rightarrow (y_1 \wedge s = 2)}_{\Delta_1},$$

we next consider the transition  $m_2$ . The formula  $\text{pre}(m_2, \Gamma_1)$  is given by

$$\begin{aligned} \forall V': \underbrace{\text{at\_}m_2 \wedge \text{at'}\_m_3 \wedge y'_1 = y_1 \wedge s' = 2 \wedge \dots}_{\rho_{m_2}} \rightarrow \\ \underbrace{\neg \text{at'}\_m_4 \wedge (\text{at'}\_m_3 \rightarrow (y'_1 \wedge s' = 2))}_{\Gamma'_1}. \end{aligned}$$

This is equivalent to

$$\Delta_2: \text{at\_}m_2 \rightarrow y_1,$$

leading to

$$\Gamma_2: \neg \text{at\_}m_4 \wedge (\text{at\_}m_3 \rightarrow s = 2) \wedge (\text{at\_}m_{2,3} \rightarrow y_1).$$

The next transitions to be considered are  $m_1$ ,  $m_0$ , and  $m_5$ , which yield the following sequence of assertions:

$$\Gamma_3: \neg \text{at\_}m_4 \wedge (\text{at\_}m_3 \rightarrow s = 2) \wedge (\text{at\_}m_{1..3} \rightarrow y_1)$$

$$\Gamma_4: \neg \text{at\_}m_4 \wedge (\text{at\_}m_3 \rightarrow s = 2) \wedge (\text{at\_}m_{0..3} \rightarrow y_1)$$

$$\Gamma_5: \neg \text{at\_}m_4 \wedge (\text{at\_}m_3 \rightarrow s = 2) \wedge (\text{at\_}m_{0..3,5} \rightarrow y_1).$$

Using the obvious control invariant  $\text{at\_}m_{0..5}$ ,  $\Gamma_5$  can be simplified to

$$\Gamma_5: \neg \text{at\_}m_4 \wedge (\text{at\_}m_3 \rightarrow s = 2) \wedge y_1.$$

Assertion  $\Gamma_5$  is not yet preserved under all nonescape transitions. In particular, it is not preserved under transition  $\ell_5$  which changes  $y_1$  from T to F. Consequently, we continue the propagation, this time via transition  $\ell_5$ .

The precondition  $\text{pre}(\ell_5, \Gamma_5)$  is given by

$$\forall V': \underbrace{\text{at\_}\ell_5 \wedge y'_1 = \text{F} \wedge \dots}_{\rho_{\ell_5}} \rightarrow \underbrace{\neg \text{at'}\_m_4 \wedge (\text{at'}\_m_3 \rightarrow s' = 2) \wedge y'_1}_{\Gamma'_5}.$$

Since  $\rho_{\ell_5}$  implies  $y'_1 = \text{F}$ ,  $\Gamma'_5$  reduces to F, and the verification condition reduces to

$$\Delta_6: \text{at\_}\ell_5 \rightarrow \text{F}.$$

We can propagate  $\Gamma_6 = \Gamma_5 \wedge \Delta_6$  via  $\ell_4$  and obtain

$$\Delta_7: \text{at\_}\ell_4 \rightarrow \text{F}.$$

Fortunately, we need not propagate via the escape transition  $\ell_3$ .

The accumulated assertion at this stage can be written as

$$\Gamma_7: \underbrace{\neg at\_m_4}_{\Gamma_0} \wedge \underbrace{(at\_m_3 \rightarrow s = 2)}_{\Delta_1, \dots, \Delta_5} \wedge y_1 \wedge \underbrace{at\_\ell_{0..3}}_{\Delta_6, \Delta_7}.$$

Using the invariant  $\varphi_1: y_1 \leftrightarrow at\_\ell_{3..5}$ , established for program MUX-PET1 in Section 1.4, assertion  $\Gamma_7$  can be simplified to

$$\Gamma_7: \neg at\_m_4 \wedge (at\_m_3 \rightarrow s = 2) \wedge at\_\ell_3.$$

This assertion can be shown to be preserved under all but the escape transition.

Note that, since the whole waiting period starts with  $P_1$  at  $\ell_3$  and ends when  $P_1$  moves to  $\ell_4$ , we could have conjoined  $q$  with  $at\_\ell_3$ , starting with

$$\tilde{\Gamma}_0: \underbrace{\neg at\_m_4}_{q} \wedge at\_\ell_3.$$

In this case the backward propagation would have produced the following sequence of iterations:

$$\tilde{\Delta}_1: (at\_m_3 \rightarrow s = 2) \wedge at\_\ell_3.$$

$$\tilde{\Gamma}_1: \neg at\_m_4 \wedge at\_\ell_3 \wedge (at\_m_3 \rightarrow s = 2).$$

at which point the process converges. ■

### Comparison of the Propagation Methods

Assume that, applying the forward and backward propagation methods to the formula  $p \Rightarrow q Wr$ , the two methods converged to the assertions  $\Phi_t$  and  $\Gamma_f$ , respectively.

In general,  $\Phi_t$  and  $\Gamma_f$  are not necessarily equivalent. For example, assertion

$$\Phi_1: (at\_\ell_3 \wedge at\_m_{0..2}) \vee (at\_\ell_3 \wedge at\_m_3 \wedge s = 2)$$

obtained by forward propagation is not equivalent to assertion  $\Gamma_7$ , which can be written as

$$\Gamma_7: (at\_\ell_3 \wedge at\_m_{0..2,5}) \vee (at\_\ell_3 \wedge at\_m_3 \wedge s = 2).$$

For example, a state in which  $\pi = \{\ell_3, m_5\}$  satisfies  $\Gamma_7$  but not  $\Phi_1$ .

This is not surprising since the two methods characterize different sets of states. Forward propagation attempts to capture in its terminal assertion  $\Phi_t$  all states that are reachable from an accessible  $p$ -state by an  $r$ -free computation segment, that is, all states  $s$  for which there exists a computation segment  $s_1, \dots, s_k = s$  such that  $s_1$  is an accessible  $p$ -state and  $s_i \not\models r$  for all  $i = 1, \dots, k$ . If the formula  $p \Rightarrow q Wr$  is  $P$ -valid then all these states must satisfy  $q$ , i.e.,  $\Phi_t \rightarrow q$  is  $P$ -state valid.

In comparison, backward propagation attempts to capture in  $\Gamma_f$  all states  $s$  such that all computations starting at  $s$  satisfy  $q \mathcal{W} r$ . If  $p \Rightarrow q \mathcal{W} r$  is  $P$ -valid then all accessible  $p$ -states must satisfy  $\Gamma_f$ , i.e.,  $p \rightarrow \Gamma_f$  is  $P$ -state valid. In fact, not only accessible  $p$ -states should satisfy  $\Gamma_f$  but also all states reachable from accessible  $p$ -states by an  $r$ -free computation segment, i.e., all  $\Phi_t$ -states. This indicates the relation between  $\Phi_t$  and  $\Gamma_f$ .

If  $p \Rightarrow q \mathcal{W} r$  is  $P$ -valid and both propagation methods converge, then

$$\Phi_t \rightarrow \Gamma_f$$

is  $P$ -state valid.

For example,  $\Phi_1$  obtained for program MUX-PET1 certainly implies  $\Gamma_7$ . States  $s$  in which  $\pi = \{\ell_3, m_5\}$  satisfy  $\Gamma_7$  and hence have the property that all computations starting at  $s$  satisfy  $(\neg at\_m_4) \mathcal{W} at\_{\ell_4}$ . On the other hand, such a state cannot be reached from a  $p$ -state, where  $p: at\_{\ell_3} \wedge at\_m_{0..2}$ , in an  $(at\_{\ell_4})$ -free computation segment, i.e., a computation segment in which control does not pass through  $\ell_4$ .

In **Problem 3.2**, the reader is requested to formulate a proof rule, similar to rule WAIT, for the *precedence* operator  $\mathcal{P}$  which, as shown in Problem 0.8, can be defined in terms of the waiting-for operator. In **Problem 3.3**, the reader is requested to use this rule for proving a precedence property.

## 3.2 Nested Waiting-for Rule

A desirable precedence property of algorithms for mutual exclusion is that of *1-bounded overtaking*, which requires that, for two processes  $P_1$  and  $P_2$ ,

From the time  $P_1$  is waiting to enter its critical section,  $P_2$  may enter its critical section ahead of  $P_1$  (overtaking  $P_1$ ) at most once.

One of the preceding examples established for program MUX-PET1 (Fig. 3.4) the property

$$\psi_2: at\_{\ell_3} \wedge at\_m_{0..2} \Rightarrow (\neg at\_m_4) \mathcal{W} at\_{\ell_4}.$$

Obviously,  $\psi_2$  implies 1-bounded overtaking for MUX-PET1. Assume that  $P_1$  were continuously waiting at  $\ell_3$  while  $P_2$  managed to visit its critical section on two separate occasions. Separating these two occasions must be a state in which  $P_1$  is at  $\ell_3$  while  $P_2$  is at  $m_1$ . However, according to  $\psi_2$ , the next visit to a critical section following such a state must be by  $P_1$  rather than by  $P_2$ . This contradicts our assumption of two successive visits to the critical section by  $P_2$  while  $P_1$  is waiting at  $\ell_3$ .

While this argument is convincing, it is still indirect. To directly formalize and then verify this property, we need nested waiting-for formulas, whose verifi-

cation will be discussed next.

For example, the requirement of 1-bounded overtaking for program MUX-PETI (Fig. 3.4) can be expressed by the formula

$$\psi_3: \quad at\text{-}\ell_3 \Rightarrow \underbrace{\neg at\text{-}m_4}_{q_3} \mathcal{W} \underbrace{at\text{-}m_4}_{q_2} \mathcal{W} \underbrace{\neg at\text{-}m_4}_{q_1} \mathcal{W} \underbrace{at\text{-}\ell_4}_{q_0}.$$

Formula  $\psi_3$  is a nested waiting-for formula.

A *nested waiting-for* formula has the form

$$p \Rightarrow q_n \mathcal{W} q_{n-1} \cdots q_1 \mathcal{W} q_0,$$

and is interpreted as the fully parenthesized formula

$$p \Rightarrow q_n \mathcal{W} (q_{n-1} \mathcal{W} (\cdots (q_1 \mathcal{W} q_0) \cdots)).$$

Recall that such a nested waiting-for formula ensures that every  $p$  is followed by a  $q_n$ -interval, followed by a  $q_{n-1}$ -interval,  $\dots$ , followed by a  $q_1$ -interval, possibly terminated by a  $q_0$ -position. Note, however, that since  $p \mathcal{W} q$  may be satisfied by a single  $q$ -position, any of these intervals may be empty. Also, due to the definition of the waiting-for operator, any of the intervals, say the  $q_i$ -interval, may extend to infinity, in which case the intervals for  $q_{i-1}, \dots, q_1$  are all empty and  $q_0$  need not occur.

Thus, formula  $\psi_3$  states that any position satisfying  $at\text{-}\ell_3$  is followed by an interval in which  $P_2$  is not at  $m_4$ , followed by an interval in which  $P_2$  is at  $m_4$ , followed by an interval in which  $P_2$  is again not at  $m_4$ , which can be terminated only by a state in which  $P_1$  is at  $\ell_4$ . It therefore implies that from the time  $P_1$  starts waiting at  $\ell_3$ , there can be at most one continuous interval in which  $P_2$  is at  $m_4$ , before  $P_1$  is admitted to  $\ell_4$ . The case of  $P_2$  not reaching  $m_4$  at all is obtained by taking an empty  $at\text{-}m_4$  interval.

## Basic Rule

The basic rule for establishing nested waiting-for properties is rule NWAIT-B, presented in Fig. 3.5. The premise of the rule requires that each transition of the program lead from an accessible state that satisfies  $\varphi_i$ , for  $i > 0$ , to a successor state that satisfies  $\varphi_j$ , for some  $j \leq i$ .

**Justification** To justify rule NWAIT-B, consider a state  $s_k$  satisfying  $\bigvee_{j=0}^m \varphi_j$ . It follows that there exists some index  $j_k$ ,  $0 \leq j_k \leq m$ , such that  $s_k$  satisfies  $\varphi_{j_k}$ .

If  $j_k = 0$  we are done, since  $\varphi_0$  terminates the required sequence of intervals.

Otherwise,  $j_k > 0$  and we consider  $s_{k+1}$ , the successor of  $s_k$ . The premise of the rule implies that  $s_{k+1}$  satisfies  $\varphi_{j_{k+1}}$  for some  $j_{k+1} \leq j_k$ . We now repeat the argument for  $s_{k+1}$ , and so on.

For assertions  $\varphi_0, \varphi_1, \dots, \varphi_m$ ,

$$\{\varphi_i\} T \left\{ \bigvee_{j \leq i} \varphi_j \right\} \text{ for } i = 1, \dots, m$$

$$\left( \bigvee_{j=0}^m \varphi_j \right) \Rightarrow \varphi_m W \varphi_{m-1} W \dots W \varphi_1 W \varphi_0$$

Fig. 3.5. Rule NWAIT-B (basic nested waiting-for).

Denoting the indices of the assertions established for  $s_k$  and its successors by  $j_k, j_{k+1}, \dots$ , the premise guarantees that the sequence of indices

$$j_k \geq j_{k+1} \geq \dots$$

can either terminate at some  $j_r = 0$  or extend to infinity. It is not difficult to see that this guarantees a sequence of intervals satisfying

$$\varphi_m, \varphi_{m-1}, \dots,$$

which may either terminate at a state satisfying  $\varphi_0$  or extend to infinity. Clearly, some of these intervals may be empty. ■

**Example** Consider the case of five assertions  $\varphi_0, \dots, \varphi_4$ . Assume that  $\bigvee_{j=0}^4 \varphi_j$  occurs at state  $s_3$ , and we have established that the states

$$s_3, s_4, s_5, s_6, s_7, s_8$$

satisfy

$$\varphi_3, \varphi_3, \varphi_3, \varphi_1, \varphi_1, \varphi_0,$$

respectively. Then, we claim that

$$\varphi_4 W \varphi_3 \dots \varphi_1 W \varphi_0$$

holds at position 3. Using the notation  $[s_i..s_j)$  to refer to the half-open interval  $s_i, \dots, s_{j-1}$ , the above claim is true because

the (empty) interval  $[s_3..s_3)$  satisfies  $\varphi_4$ , and is followed by

the interval  $[s_3..s_6)$  satisfying  $\varphi_3$ , which is followed by

the (empty) interval  $[s_6..s_6)$  satisfying  $\varphi_2$ , which is followed by

the interval  $[s_6..s_8)$  satisfying  $\varphi_1$ , which is followed by

a  $\varphi_0$ -position at  $s_8$ . ■

## General Rule

Rule NWAIT-B is not always directly applicable. As expected, in many cases we must first strengthen some of the participating assertions. This is supported by the more general version of the rule, called the nested waiting-for rule (NWAIT), and presented in Fig. 3.6.

For assertions  $p, q_0, q_1, \dots, q_m$ , and  $\varphi_0, \varphi_1, \dots, \varphi_m$ ,

$$\text{N1. } p \rightarrow \bigvee_{j=0}^m \varphi_j$$

$$\text{N2. } \varphi_i \rightarrow q_i \quad \text{for } i = 0, 1, \dots, m$$

$$\text{N3. } \{\varphi_i\} T \left\{ \bigvee_{j \leq i} \varphi_j \right\} \quad \text{for } i = 1, \dots, m$$

$$p \Rightarrow q_m W q_{m-1} \dots q_1 W q_0$$

Fig. 3.6. Rule NWAIT (nested waiting-for).

**Justification** To justify this rule, consider a state  $s_k$  satisfying  $p$ . By premise N1, state  $s_k$  also satisfies  $\bigvee_{j=0}^m \varphi_j$ . Due to premise N3 and rule NWAIT-B, this guarantees a sequence of intervals satisfying, respectively,  $\varphi_m, \varphi_{m-1}, \dots$  which either terminates at a state satisfying  $\varphi_0$  or extends to infinity. By premise N2, any interval or state satisfying  $\varphi_i$  also satisfies  $q_i$ . It follows that  $q_m W q_{m-1} \dots q_1 W q_0$  holds at position  $k$  in the computation. ■

**Problem 3.4** asks the reader to show that every nested waiting-for formula  $p \Rightarrow q_m W q_{m-1} \dots q_1 W q_0$  is equivalent to a nested waiting-for formula of the form

$$T \Rightarrow (\neg p) W q_m W q_{m-1} \dots q_1 W q_0.$$

**Example** (Peterson's algorithm — version 2)

As an example of an application of the general nested waiting-for rule, consider again Peterson's algorithm for implementing mutual exclusion. This time we prefer to consider the refined version MUX-PET2 of Fig. 3.7.

In Section 1.4 (Fig. 1.14, page 124), we established the following invariants for this program:

**local**  $y_1, y_2$ : boolean where  $y_1 = F, y_2 = F$   
 $s$  : integer where  $s = 1$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: y_1 := T \\ \ell_3: s := 1 \\ \ell_4: \left[ \begin{array}{l} \ell_4^a: \text{await } \neg y_2 \\ \text{or} \\ \ell_4^b: \text{await } s \neq 1 \end{array} \right] \\ \ell_5: \text{critical} \\ \ell_6: y_1 := F \end{array} \right] \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: y_2 := T \\ m_3: s := 2 \\ m_4: \left[ \begin{array}{l} m_4^a: \text{await } \neg y_1 \\ \text{or} \\ m_4^b: \text{await } s \neq 2 \end{array} \right] \\ m_5: \text{critical} \\ m_6: y_2 := F \end{array} \right] \end{array} \right]$

Fig. 3.7. Program MUX-PET2 (Peterson's algorithm) — version 2.

- $\chi_0: s = 1 \vee s = 2$
- $\chi_1: y_1 \leftrightarrow at_{-}\ell_{3..6}$
- $\chi_2: y_2 \leftrightarrow at_{-}m_{3..6}$
- $\chi_3: at_{-}\ell_4 \wedge at_{-}m_5 \rightarrow y_2 \wedge s = 1$
- $\chi_4: at_{-}\ell_5 \wedge at_{-}m_4 \rightarrow y_1 \wedge s = 2.$

We wish to prove that there is a *1-bounded overtaking* from the time a process starts waiting to enter its critical section. This is expressed by the nested waiting-

for formula:

$$\psi: \underbrace{at_{-\ell_4}}_p \Rightarrow \underbrace{\neg at_{-m_5}}_{q_3} \mathcal{W} \underbrace{at_{-m_5}}_{q_2} \mathcal{W} \underbrace{\neg at_{-m_5}}_{q_1} \mathcal{W} \underbrace{at_{-\ell_5}}_{q_0}.$$

To use rule NWAIT we have to find four assertions  $\varphi_0, \varphi_1, \varphi_2, \varphi_3$ , whose disjunction is implied by  $at_{-\ell_4}$  (satisfying N1), which strengthen assertions  $at_{-\ell_5}, \neg at_{-m_5}, at_{-m_5}, \neg at_{-m_5}$ , respectively (satisfying N2), and which satisfy the verification conditions of premise N3 of the rule.

A natural candidate for  $\varphi_0$  is  $at_{-\ell_5}$  itself,

$$\varphi_0: at_{-\ell_5},$$

because obviously it terminates the waiting period. Proceeding to  $\varphi_1$ , assertion  $\varphi_1$  should strengthen  $\neg at_{-m_5}$ . We can safely conjoin it with  $at_{-\ell_4}$ , since the whole period starts with  $P_1$  at  $\ell_4$  and terminates when  $P_1$  moves to  $\ell_5$ .

What additional information should we include in  $\varphi_1$ ? Considering the role of  $\varphi_1$  in the waiting-for formula and premise N3, the only exit from a  $\varphi_1$ -state to a  $(\neg \varphi_1)$ -state should be to an  $(at_{-\ell_5})$ -state. It follows that  $\varphi_1$  should characterize all the states in which the next entry to a critical section will be by  $P_1$ , i.e., all the states in which  $P_1$  has a definite priority over  $P_2$ .

Performing a backward propagation from

$$\Psi_0: at_{-\ell_4} \wedge \neg at_{-m_5}$$

via  $m_4$ , we obtain the supplementing assertion

$$\Phi_1: at_{-m_4} \rightarrow y_1 \wedge s = 2.$$

Taking the conjunction  $\Psi_0 \wedge \Phi_1$ , and dropping the conjunct  $y_1$ , which is implied (due to  $\chi_1$ ) by  $at_{-\ell_4}$ , we obtain an assertion that can also be written as

$$\varphi_1: at_{-\ell_4} \wedge \left( at_{-m_{0..3,6}} \vee (at_{-m_4} \wedge s = 2) \right).$$

For assertion  $\varphi_2$ , it seems sufficient to take

$$\varphi_2: at_{-\ell_4} \wedge at_{-m_5}.$$

For  $\varphi_3$ , we have to characterize all the states in which  $P_2$  has priority over  $P_1$ , while  $P_1$  waits at  $\ell_4$ . Seeing that  $\varphi_1$  and  $\varphi_2$  cover almost all the configurations satisfying  $at_{-\ell_4}$ , the only remaining one is given by

$$\varphi_3: at_{-\ell_4} \wedge at_{-m_4} \wedge s = 1.$$

From the way  $\varphi_0$ - $\varphi_3$  were constructed, it is obvious that  $at_{-\ell_4}$  implies their disjunction, that each of them is a strengthening of the corresponding assertion in the nested waiting-for specification, and that  $\varphi_1$ - $\varphi_3$  satisfy premise N3 of rule NWAIT.

It therefore follows that the following 1-bounded overtaking property is  $P$ -

valid:

$$\psi: \text{at\_}\ell_4 \Rightarrow (\neg \text{at\_}\ell_5) \mathcal{W} \text{at\_}\ell_5 \mathcal{W} (\neg \text{at\_}\ell_5) \mathcal{W} \text{at\_}\ell_5.$$

## Properties of Waiting-for Formulas

There are several useful properties of nested waiting-for formulas that prove useful. Below, we list two of them as rules.

The first rule expresses a *concatenation property* of nested waiting-for formulas:

Rule CONC-W (concatenation of waiting-for formulas)

$$\frac{p \Rightarrow q_m \mathcal{W} \dots q_1 \mathcal{W} q_0 \quad q_0 \Rightarrow r_n \mathcal{W} \dots \mathcal{W} r_0}{p \Rightarrow q_m \mathcal{W} \dots \mathcal{W} q_1 \mathcal{W} r_n \mathcal{W} \dots \mathcal{W} r_0}$$

Consider a state  $s_i$  in a computation, satisfying  $p$ . According to the first premise,  $s_i$  starts a sequence of intervals satisfying  $q_m, q_{m-1}, \dots, q_1$ , and possibly terminated by a  $q_0$ -state. If the  $q_0$ -state does occur, then the second premise guarantees a continuing sequence of intervals satisfying  $r_n, r_{n-1}, \dots, r_1$ , possibly terminated by  $r_0$ . This shows that the nested waiting-for formula in the rule's consequence holds at position  $i$ . In the case that one of the  $q_j$ -intervals extends to infinity, the concatenated nested waiting-for formula is also satisfied at position  $i$ .

We now present another rule, the *collapsing property*, of nested waiting-for formulas.

Rule COLL-W (collapsing of waiting-for formulas)

For  $i > 0$

$$\frac{p \Rightarrow q_m \mathcal{W} \dots \mathcal{W} q_{i+1} \mathcal{W} q_i \mathcal{W} \dots \mathcal{W} q_0}{p \Rightarrow q_m \mathcal{W} \dots \mathcal{W} (q_{i+1} \vee q_i) \mathcal{W} \dots \mathcal{W} q_0}$$

This rule allows us to collapse two consecutive intervals, a  $q_{i+1}$ -interval and a  $q_i$ -interval into a single  $(q_{i+1} \vee q_i)$ -interval, as specified by the conclusion. The soundness of this rule is obvious.

**Example** (Peterson's algorithm — version 2)

In the previous example, we chose to express the 1-bounded overtaking properties

of program MUX-PET2 of Fig. 3.7, starting at  $\ell_4$ . One may wonder what overtaking can occur from the time  $P_1$  arrives at  $\ell_3$ . It is possible to show that from  $\ell_3$  we can only guarantee a bound of 2 on the number of overtakings. The property of 2-bounded overtaking from  $\ell_3$  is stated by the formula

$$\psi: \text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \\ \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5.$$

We can prove this property directly by rule NWAIT. However, we prefer to prove this property while illustrating additional techniques for handling nested waiting-for formulas.

First we prove that while  $P_1$  is on its way from  $\ell_3$  to  $\ell_4$ ,  $P_2$  can be in its critical section at most once. This property of 1-bounded overtaking from  $\ell_3$  is stated by the formula:

$$\psi_1: \text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_4.$$

The proof is a simple adaptation of the proof of the corresponding formula for 1-bounded overtaking from  $\ell_4$ . We take  $\hat{\varphi}_0$ - $\hat{\varphi}_3$  to be identical to the assertions  $\varphi_0$ - $\varphi_3$  used there, systematically replacing occurrences of  $\text{at\_}\ell_5$  by  $\text{at\_}\ell_4$  and occurrences of  $\text{at\_}\ell_4$  by  $\text{at\_}\ell_3$ .

In the previous example, we proved the property of 1-bounded overtaking from  $\ell_4$ , stated by the formula:

$$\psi_2: \text{at\_}\ell_4 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5.$$

Applying the concatenation rule CONC-W for nested waiting-for formulas, we concatenate  $\psi_1$  and  $\psi_2$  to obtain

$$\text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \\ (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5.$$

We now use the collapsing rule COLL-W on the formula above to merge the two consecutive  $\neg\text{at\_}m_5$  intervals. Thus, we conclude

$$\psi: \text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \\ \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5,$$

which precisely expresses 2-bounded overtaking starting at  $\ell_3$ .

Encouraged by this result, we may ask what bound exists on the overtaking, starting from  $\ell_2$ . Unfortunately, there is no such bound. For any given  $n \geq 0$ , we can present a computation in which, from the time  $P_1$  is at  $\ell_2$ ,  $P_2$  enters  $m_5$  successively  $n$  times before  $P_1$  ever gets to  $\ell_5$ . ■

In **Problems 3.5–3.8**, the reader is requested to analyze the overtaking properties of some other programs proposed as solutions of the mutual-exclusion problem.

### 3.3 Verification Diagrams

In proofs such as we have been conducting, it is typically necessary to deal with several assertions at the same time and trace which transitions lead from one assertion to another. It is convenient to visualize this situation by a diagram that summarizes the assertions under consideration and the possible transitions between them. Since a diagram provides a succinct representation of a large set of verification conditions, it can often present a useful and illuminating overview of a complex proof.

We define a *verification diagram* to be a directed labeled graph constructed as follows:

- *Nodes* in the graph are labeled by assertions. We will often refer to a node by the assertion labeling it.
- *Edges* in the graph represent transitions between assertions. Each edge departs from one assertion, connects to another, and is labeled by the name of a transition in the program. We refer to an edge labeled by  $\tau$  as a  $\tau$ -edge.
- One of the nodes may be designated as a *terminal node* (“goal” node). In our graphical representation, this node is distinguished by having a boldface boundary. No edges depart from a terminal node. Terminal nodes correspond to “goal” assertions such as  $\varphi_0$  in a waiting-for formula.

Verification diagrams provide a concise representation of sets of verification conditions as follows:

For a nonterminal  $\varphi$ -node and transition  $\tau$ , let  $\varphi_1, \dots, \varphi_k$  be the nodes reached by  $\tau$ -edges departing from  $\varphi$ . The *verification condition* associated with  $\varphi$  and  $\tau$  is given by

$$\{\varphi\} \tau \{\varphi \vee \varphi_1 \vee \cdots \vee \varphi_k\}.$$

Note that for the case  $k = 0$ , i.e., no  $\tau$ -edges depart from  $\varphi$ , the verification condition for  $\varphi$  and  $\tau$  is given by

$$\{\varphi\} \tau \{\varphi\}.$$

We describe the situation that no  $\tau$ -edges depart from assertion  $\varphi$  by saying that there is an *implicit*  $\tau$ -edge connecting  $\varphi$  to itself. No verification conditions are associated with terminal nodes.

We define a verification diagram to be *valid over a program P* (*P-valid* for

short) if all the verification conditions associated with nodes of the diagram are  $P$ -state valid.

A valid diagram provides a succinct representation of all the verification conditions of the form  $\{\varphi\} \tau \{\psi\}$ , for all combinations of assertions  $\varphi$  and  $\psi$  and transitions  $\tau$  that appear in the diagram. Consequently, a diagram can present a most useful and illuminating overview of a complex proof, establishing the premises of a rule such as rule NWAIT.

## Wait Diagrams

A verification diagram with nodes  $\varphi_m, \dots, \varphi_0$  is called a *WAIT diagram* if  $\varphi_0$  is a terminal node and the diagram is weakly acyclic, i.e., whenever node  $\varphi_i$  is connected by an edge to node  $\varphi_j$ , then  $i \geq j$ .  $P$ -valid WAIT diagrams can be used to establish the  $P$ -validity of a nested waiting-for formula.

The consequences of having a  $P$ -valid WAIT diagram are stated by the following claim:

**Claim 3.1** (WAIT diagram)

A  $P$ -valid WAIT diagram establishes that the formula

$$\bigvee_{j=0}^m \varphi_j \Rightarrow \varphi_m \mathcal{W} \varphi_{m-1} \cdots \varphi_1 \mathcal{W} \varphi_0$$

is  $P$ -valid.

If, in addition, we can establish the  $P$ -state validity of the following implications:

$$(N1) \quad p \rightarrow \bigvee_{j=0}^m \varphi_j \quad \text{and} \quad (N2) \quad \varphi_i \rightarrow q_i \quad \text{for } i = 0, 1, \dots, m,$$

then, we can conclude the  $P$ -validity of

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0.$$

**Justification** The first part of the claim, showing that a  $P$ -valid WAIT diagram implies the  $P$ -validity of

$$\bigvee_{j=0}^m \varphi_j \Rightarrow \varphi_m \mathcal{W} \varphi_{m-1} \cdots \varphi_1 \mathcal{W} \varphi_0$$

follows directly from rule NWAIT by taking

$$p = \bigvee_{j=0}^m \varphi_j \quad \text{and} \quad q_i = \varphi_i \text{ for each } i = 0, \dots, m.$$

Premises N1 and N2 are trivial due to the choice of  $p$  and  $q_0, \dots, q_m$ . Premise N3 follows from the verification conditions associated with the WAIT diagram.

The second part of the claim is even simpler, because premises N1 and N2 are explicitly ensured. ■

As seen in the preceding discussion, WAIT diagrams provide a graphical presentation of assertions  $\varphi_0, \dots, \varphi_m$  satisfying premise N3 of rule NWAIT. That is,  $\varphi_0, \dots, \varphi_m$  can be included in a  $P$ -valid WAIT diagram iff they satisfy premise N3.

### Example (Peterson's algorithm — version 2)

Consider program MUX-PET2 (Fig. 3.7). In Fig. 3.8 we present a WAIT diagram containing the assertions  $\varphi_0 \dots \varphi_3$  that were used to prove the property of 1-bounded overtaking from  $\ell_4$ . This property was expressed by the nested waiting-for formula

$$\psi: \underbrace{at_{-\ell_4}}_p \Rightarrow \underbrace{\neg at_{-m_5}}_{q_3} \mathcal{W} \underbrace{at_{-m_5}}_{q_2} \mathcal{W} \underbrace{\neg at_{-m_5}}_{q_1} \mathcal{W} \underbrace{at_{-\ell_5}}_{q_0}.$$

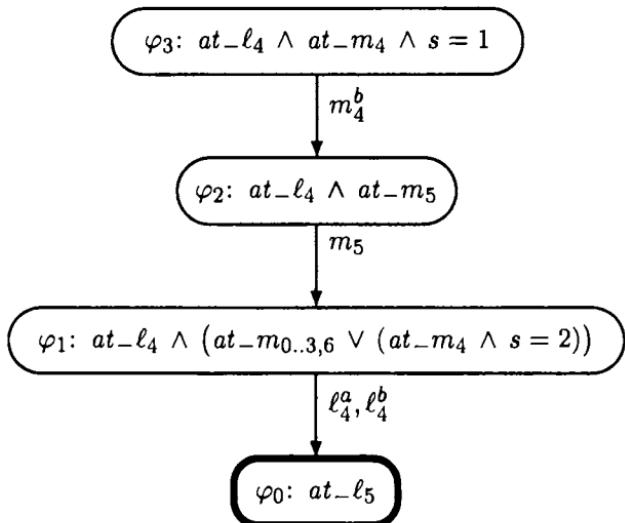


Fig. 3.8. WAIT diagram for 1-bounded overtaking from  $\ell_4$ .

The WAIT diagram of Fig. 3.8 states the validity of the following verification conditions, in which we use the notation  $\overline{m_i}$  to denote the set of all transitions in  $\mathcal{T}$  different from  $m_i$ , and similarly for  $\overline{\ell_j}$ .

- Conditions from  $\varphi_3$

$$\{\varphi_3\} m_4^b \{\varphi_3 \vee \varphi_2\} \quad \{\varphi_3\} \overline{m_4^b} \{\varphi_3\}.$$

- Conditions from  $\varphi_2$

$$\{\varphi_2\} m_5 \{\varphi_2 \vee \varphi_1\} \quad \{\varphi_2\} \overline{m_5} \{\varphi_2\}.$$

- Conditions from  $\varphi_1$

$$\{\varphi_1\} \ell_4^a \{\varphi_1 \vee \varphi_0\} \quad \{\varphi_1\} \ell_4^b \{\varphi_1 \vee \varphi_0\} \quad \{\varphi_1\} \{\overline{\ell_4^a, \ell_4^b}\} \{\varphi_1\}.$$

These verification conditions are not generally state valid. However, they are  $P$ -state valid for program MUX-PET2. This can be proven by showing that each of them is implied by the conjunction of the invariants  $\chi_0$ – $\chi_4$  listed (page 268) in the proof of the nested waiting-for property  $\psi$ .

Consequently, the WAIT diagram of Fig. 3.8 is valid over program MUX-PET2.

Also, the implications

$$\begin{aligned} \underbrace{at_{-\ell_4}}_p &\rightarrow \bigvee_{j=0}^3 \varphi_j, \\ \varphi_0 &\rightarrow \underbrace{at_{-\ell_5}}_{q_0}, \quad \varphi_1 \rightarrow \underbrace{\neg at_{-m_5}}_{q_1}, \\ \varphi_2 &\rightarrow \underbrace{at_{-m_5}}_{q_2}, \quad \varphi_3 \rightarrow \underbrace{\neg at_{-m_5}}_{q_3} \end{aligned}$$

are  $P$ -state valid. Consequently, the diagram establishes the validity of the formula

$$at_{-\ell_4} \Rightarrow (\neg at_{-m_5}) \mathcal{W} at_{-m_5} \mathcal{W} (\neg at_{-m_5}) \mathcal{W} at_{-\ell_5}$$

over program MUX-PET2. ■

## Invariance Diagrams

Another class of verification diagrams can be used to establish invariance properties.

A verification diagram is called an **INVARIENCE diagram** if it is exit-free, i.e., it has no terminal node. Unlike WAIT diagrams, INVARIENCE diagrams may contain cycles.

### Claim 3.2 (INVARIENCE diagram)

A  $P$ -valid INVARIENCE diagram with nodes  $\varphi_1, \dots, \varphi_m$  establishes that the formula

$$\bigvee_{j=1}^m \varphi_j \Rightarrow \square \left( \bigvee_{j=1}^m \varphi_j \right)$$

is  $P$ -valid.

If, in addition, we can establish the  $P$ -state validity of the implications

$$(I1) \quad \bigvee_{j=1}^m \varphi_j \rightarrow q \quad \text{and} \quad (I2) \quad \Theta \rightarrow \bigvee_{j=1}^m \varphi_j$$

then, we can conclude the  $P$ -validity of

$$\square q.$$

**Justification** Denote by  $\varphi$  the disjunction

$$\varphi: \bigvee_{j=1}^m \varphi_j.$$

Consider an assertion  $\varphi_i$ ,  $1 \leq i \leq m$ , and a transition  $\tau$ . Let  $\varphi_{j_1}, \dots, \varphi_{j_t}$ ,  $j_1 \dots j_t \in [1..m]$ , be all the  $\tau$ -successors of  $\varphi_i$ , i.e., the assertions to which  $\varphi_i$  is connected by a  $\tau$ -labeled edge. Since the diagram is  $P$ -valid, we know that the verification condition

$$\{\varphi_i\} \tau \{\varphi_i \vee \varphi_{j_1} \vee \dots \vee \varphi_{j_t}\}: \rho_\tau \wedge \varphi_i \rightarrow \varphi'_i \vee \varphi'_{j_1} \vee \dots \vee \varphi'_{j_t}$$

is  $P$ -state valid. Obviously this implies the weaker implication

$$\{\varphi_i\} \tau \{\varphi\}: \rho_\tau \wedge \varphi_i \rightarrow \varphi'_1 \vee \dots \vee \varphi'_m.$$

Since this was derived for an arbitrary  $i$ ,  $1 \leq i \leq m$ , we can take the disjunction of the implications for all  $i = 1, \dots, m$ . This leads to

$$\{\varphi\} \tau \{\varphi\}: \rho_\tau \wedge (\varphi_1 \vee \dots \vee \varphi_m) \rightarrow \varphi'_1 \vee \dots \vee \varphi'_m.$$

Since this was derived for an arbitrary  $\tau \in \mathcal{T}$ , we can take the conjunction of the verification conditions for all  $\tau \in \mathcal{T}$  and conclude

$$\{\varphi\} \mathcal{T} \{\varphi\}.$$

Consider a computation  $\sigma$  and position  $k \geq 0$  such that  $\varphi$  holds at  $k$ . Clearly,  $\{\varphi\} \mathcal{T} \{\varphi\}$  implies that  $\varphi$  holds also at position  $k + 1$ , and then at positions  $k + 2, k + 3, \dots$ . We conclude that  $\square \varphi$  holds at position  $k$ . This establishes the  $P$ -validity of

$$\varphi \Rightarrow \square \varphi.$$

The second part of the claim follows by monotonicity. ■

**Example** (Peterson's algorithm — version 2)

Let us establish that

$$\chi_1: y_1 \leftrightarrow \text{at\_}\ell_{3..6}$$

is an invariant of program MUX-PET2 (Fig. 3.7).

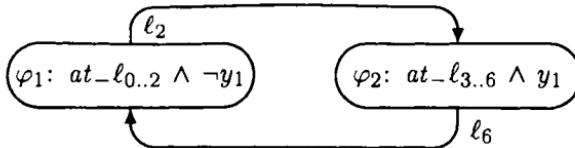


Fig. 3.9. INVARIANCE diagram.

Consider the INVARIANCE diagram presented in Fig. 3.9. It is not difficult to ascertain that this diagram is valid for program MUX-PET2.

Condition I2, monotonicity with respect to  $\Theta$ , follows from the fact that

$$\underbrace{at_{-\ell_0} \wedge \neg y_1 \wedge \dots}_{\Theta} \rightarrow \underbrace{at_{-\ell_{0..2}} \wedge \neg y_1}_{\varphi_1} \vee \underbrace{\dots}_{\varphi_2}.$$

It is straightforward to establish condition I1, monotonicity with respect to  $q$  (assertion  $\chi$ , in this case), observing that each of the assertions implies  $q$ :

$$\begin{aligned} \underbrace{at_{-\ell_{0..2}} \wedge \neg y_1}_{\varphi_1} &\rightarrow \underbrace{y_1 \leftrightarrow at_{-\ell_{3..6}}}_{q} \\ \underbrace{at_{-\ell_{3..6}} \wedge y_1}_{\varphi_2} &\rightarrow \underbrace{y_1 \leftrightarrow at_{-\ell_{3..6}}}_{q}. \end{aligned}$$

We conclude that the assertion

$$y_1 \leftrightarrow at_{-\ell_{3..6}}$$

is an invariant of program MUX-PET2. ■

## Compound Nodes

We introduce some conventions that can be described as encapsulation conventions. They lead to more structured, hierarchical diagrams and improve the readability and manageability of large complex diagrams.

The basic construct of encapsulation is the introduction of a *compound node* containing internal nodes. We refer to the contained nodes as the *descendants* of the compound node. Nodes that are not compound are called *basic nodes*.

- *Departing edges*

An edge departing from a compound node is interpreted as though it departed from each of its descendants. We thus have the graphical equivalence of Fig. 3.10.

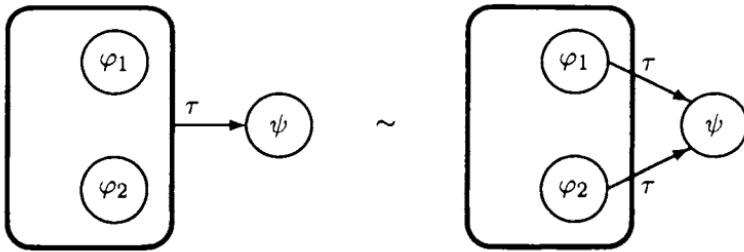


Fig. 3.10. Departing edges.

- *Arriving edges*

In a similar way, an edge arriving at a compound node is interpreted as though it arrived at each of its descendants. Thus, we have the graphical equivalence of Fig. 3.11.

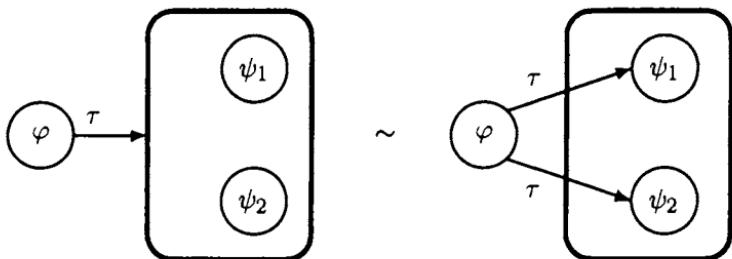


Fig. 3.11. Arriving edges.

- *Common factors*

An assertion  $\varphi$  labeling a compound node is interpreted as though it were a conjunct added to each of its descendants. This is described by the graphical equivalence presented in Fig. 3.12.

### **Example** (Peterson's algorithm — version 2)

As an illustration of the encapsulation conventions, we present in Fig. 3.13 a WAIT diagram that presents the information of Fig. 3.8 in a different form.

Both diagrams are valid and establish the same 1-bounded overtaking property. However, the diagram of Fig. 3.13 contains more information than that of Fig. 3.8, and may provide more insight into the reasons this property holds. It indicates that, within the compound node  $\varphi_1$ , the program can only progress from  $at\_m_{0..3,6}$  to  $at\_m_4 \wedge s = 2$ , but not vice versa. ■

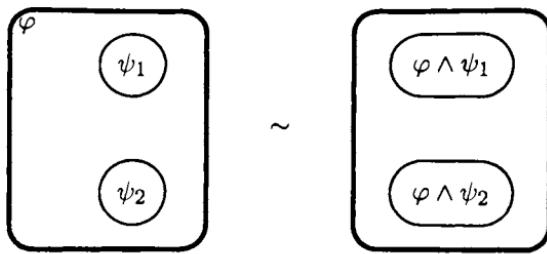


Fig. 3.12. Common factors.

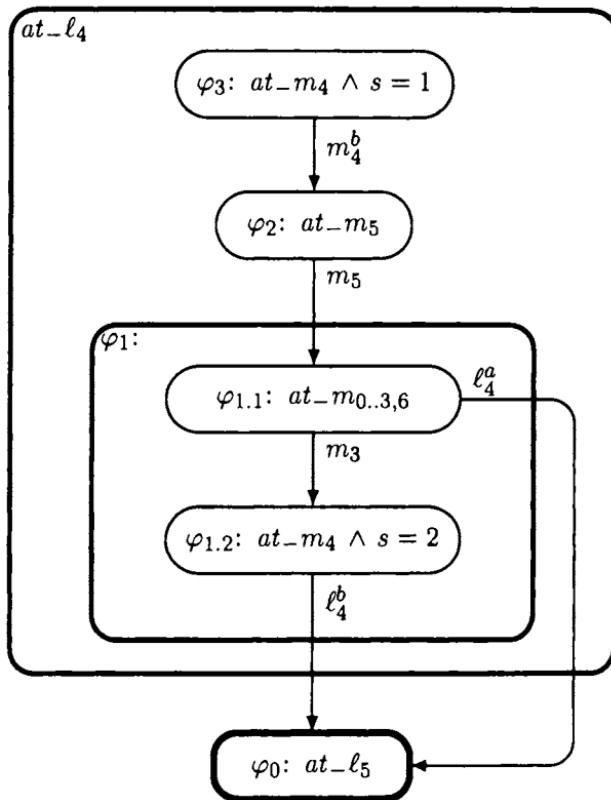


Fig. 3.13. Structured WAIT diagram for 1-bounded overtaking.

### Additional Precedence Properties

On page 271, we analyzed the overtaking situation from  $\ell_3$  and concluded that

the number of overtakings is bounded by 2, expressed by the formula

$$\psi: \text{at\_}\ell_3 \Rightarrow (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \\ \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5.$$

By conducting a more thorough analysis we can conclude that, actually, the overtaking bound from  $\ell_3$  is  $1\frac{1}{2}$ . This means that, at worst, there could be two exits from and one entry into  $m_{5,6}$ . This property is stated by the formula

$$\psi_{1.5}: \text{at\_}\ell_3 \Rightarrow \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}m_5 \mathcal{W} (\neg\text{at\_}m_5) \mathcal{W} \text{at\_}\ell_5.$$

The diagram in Fig. 3.14 provides a proof of  $1\frac{1}{2}$ -bounded overtaking from  $\ell_3$ . The proof by this diagram relies on the invariants  $\chi_0$ ,  $\chi_1$ , and  $\chi_2$ .

In Problem 3.9 and Problem 3.10, we ask the reader to construct verification diagrams for proving invariance and waiting-for properties of some programs.

**Problem 3.11** requests the reader to analyze the overtaking properties of Program MUX-SWAP.

### 3.4 Overtaking Analysis for a Resource Allocator

We conclude the discussion of waiting-for properties by giving a final example of a parameterized program which we analyze for invariance and precedence properties.

Program RES-MP, shown in Fig. 3.15, presents a solution to the distribution of a single resource between several customers that compete for it.

This program was first introduced in Section 2.2 (see Fig. 2.8, page 182), where we established the property of mutual exclusion for it which can be stated by the invariance of the assertion

$$N_3 \leq 1,$$

or equivalently

$$\neg(\text{at\_}\ell_3[k] \wedge \text{at\_}\ell_3[n]) \quad \text{for } k, n \in [1..M], k \neq n.$$

Mutual exclusion is implied by the inductive assertion

$$\chi[i]: \text{at\_}\ell_{3,4}[i] \leftrightarrow \text{at\_}m_2 \wedge t = i.$$

Note that it is up to us whether we wish to consider the critical section as consisting of location  $\ell_3$  alone, or as comprising the more extensive range  $\{\ell_3, \ell_4\}$ . Since the movement from  $\ell_3$  to  $\ell_4$  cannot be observed by any external process, most of the claims holding for  $\ell_3$  alone also hold for the extended critical section  $\{\ell_3, \ell_4\}$ . For example,  $\chi[i]$ , which is stated for  $\ell_{3,4}$ , can be used to infer mutual

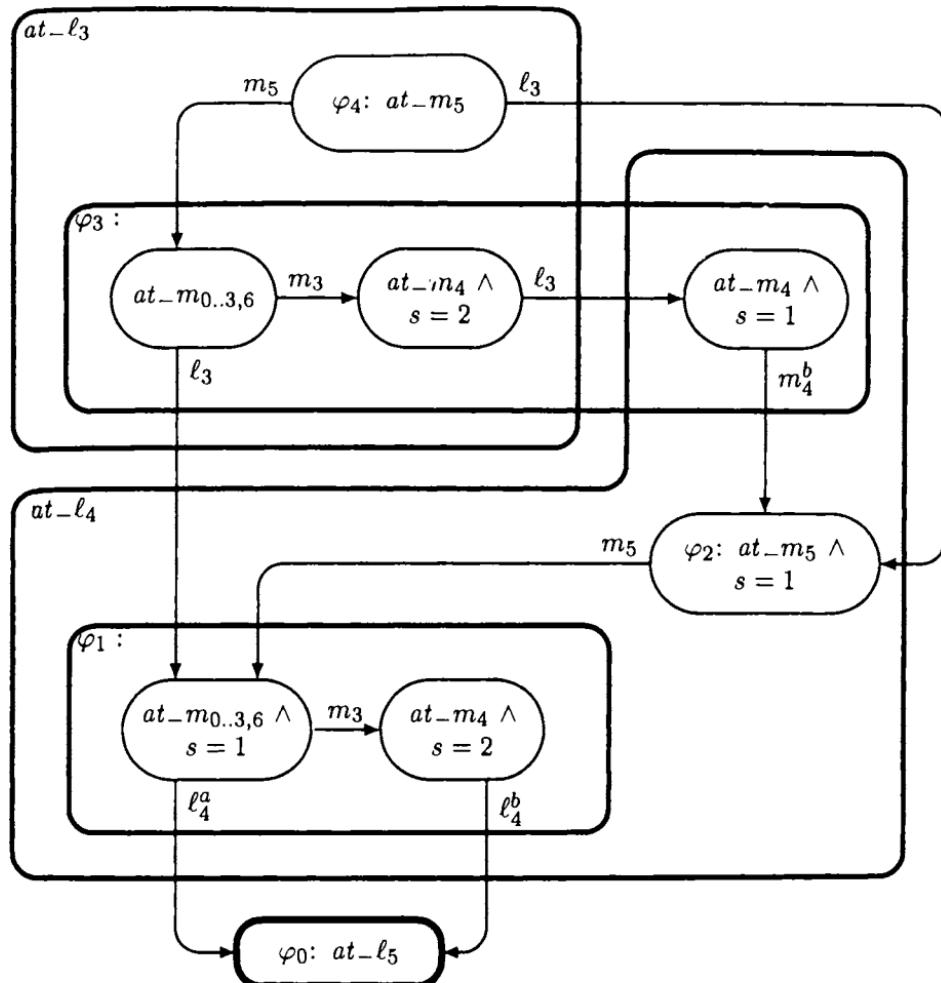


Fig. 3.14. Proving  $1\frac{1}{2}$ -bounded overtaking from  $\ell_3$ .

exclusion for the extended critical section, which can be expressed by either

$$N_{3,4} \leq 1,$$

or equivalently

$$\neg(at_{\ell_{3,4}}[k] \wedge at_{\ell_{3,4}}[n]) \quad \text{for } k, n \in [1..M], k \neq n.$$

## Overtaking Analysis

The property we wish to verify here for program RES-MP is that of bounded overtaking. As usual, it falls short of a strict first-come-first-served policy because of

in  $M$ : integer where  $M > 0$   
 local  $\alpha$  : array [1.. $M$ ] of integer channel

$A :: \left[ \begin{array}{l} \text{local } d, t: \text{integer where } t = 1 \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \text{prefer} \\ \left[ \begin{array}{l} m_1^a: \alpha[t] \Rightarrow d \\ m_2: (\alpha[t] \Rightarrow d; t := t \oplus_M 1) \end{array} \right] \\ \text{to} \\ m_1^b: t := t \oplus_M 1 \end{array} \right] \end{array} \right]$

||

$\prod_{i=1}^M C[i] :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: \alpha[i] \Leftarrow 1 \\ \ell_3: \text{critical} \\ \ell_4: \alpha[i] \Leftarrow 0 \end{array} \right] \end{array} \right]$

Fig. 3.15. Program RES-MP (resource allocator) —  
 synchronous message-passing version.

the inherent delay in detecting a request. For example, the following three events may happen in succession:

- $A$  sets  $t$  to 2 in  $m_1^b$  and moves to  $m_1$ ,
- $C[1]$  moves to  $\ell_2$ ,
- $C[2]$  moves to  $\ell_2$ .

Since the request by  $C[1]$  was made *after*  $A$  attempted to communicate with  $C[1]$ ,  $C[2]$  will be served before  $C[1]$ , even though  $C[1]$  moved to  $\ell_2$  before  $C[2]$  did.

The best we can hope for is proving 1-bounded overtaking. This property states that, from the time  $C[k]$  makes a request (by moving to  $\ell_2$ ),  $C[n]$ ,  $n \neq k$ , may precede  $C[k]$  in owning the resource at most once.

The 1-bounded overtaking we wish to prove can be expressed by

$$\psi: \underbrace{at\_l_2[k]}_p \Rightarrow \underbrace{\neg at\_l_{3,4}[n]}_{q_3} \mathcal{W} \underbrace{at\_l_{3,4}[n]}_{q_2} \mathcal{W} \underbrace{\neg at\_l_{3,4}[n]}_{q_1} \mathcal{W} \underbrace{at\_l_{3,4}[k]}_{q_0}$$

for every  $k, n \in [1..M]$ ,  $k \neq n$ .

We fix our attention on some  $k$  and  $n$  such that  $k \neq n$  and proceed to present the proof.

Following our standard heuristic for using rule NWAIT, we try to identify assertions  $\varphi_0, \dots, \varphi_3$ , such that they form an inductive chain, that is,

$$N3. \quad \{\varphi_i\} T \left\{ \bigvee_{j \leq i} \varphi_j \right\} \quad \text{for } i = 1, \dots, 3,$$

and satisfy

$$N1. \quad p \rightarrow \bigvee_{j=0}^3 \varphi_j$$

$$N2. \quad \varphi_i \rightarrow q_i \quad \text{for } i = 0, \dots, 3.$$

The search for assertions  $\varphi_0, \dots, \varphi_3$  attempts to partition the space of all states satisfying  $p$ :  $at\_l_2[k]$  or  $q$ :  $at\_l_{3,4}[k]$  into four sets, where each assertion  $\varphi_i$ ,  $i = 0, \dots, 3$ , characterizes the set of all states that satisfy

$$q_i \mathcal{W} q_{i-1} \mathcal{W} \dots \mathcal{W} q_0.$$

A reasonable choice of parameters for defining this partition consists of the location of  $A$  and the current value of  $t$ . For example, if  $t \neq n$ , then clearly  $A$  does not currently pay any attention to  $C[n]$ . If  $t = n$ , we can still analyze the situation in terms of the location of  $A$ . For example, we can partition the states in which  $t = n$  into the two sets

*Pre-Critical:*  $at\_m_{0,1}$  holds

*Critical:*  $at\_m_2$  holds

The *pre-critical* states are those in which  $A$  may still grant  $C[n]$  the resource in the current round, i.e., before changing  $t$ . The *critical* states are those in which  $C[n]$  owns the resource.

- *Constructing  $\varphi_0$*

Assertion  $\varphi_0$  should characterize the states in which  $C[k]$  already owns the resource. The simplest choice for  $\varphi_0$  is

$$\varphi_0: t = k \wedge at\_m_2,$$

which is equivalent to  $q_0$  by invariant  $\chi[k]$ .

- *Constructing  $\varphi_1$*

Next, we define  $\varphi_1$ , which should characterize all the states in which  $C[k]$  does not yet own the resource, but will own it before  $C[n]$  does. Essentially, these are all the states in which  $A$  has passed the point of scanning or granting the resource to process  $C[n]$ , but has not yet granted the resource to process  $C[k]$ . There are two cases covering this situation:

- $t$  is after  $n$  and before  $k$

To characterize this case, we introduce the notation

$$\delta(i, j) = (j - i) \bmod M,$$

called the *cyclic distance* between  $i$  and  $j$ . The function  $\delta(i, j)$  measures the distance, in cyclic increasing order, from  $i$  to  $j$ . In the context considered here, assume that the current value of  $t$  is  $i$ . Then  $\delta(i, j)$  measures the number of times  $t$  will have to be cyclically incremented before it reaches the value  $j$ .

For example, if  $i = j$  then  $\delta(i, j) = 0$  implying that no further increments are necessary before  $t$  assumes the value  $j$ .

As another example, consider the case  $i = 2$  and  $j = 1$ . For this case we have  $\delta(2, 1) = (-1) \bmod M = M - 1$ , meaning that starting at 2, and progressing through  $3, \dots, M, 1$ , we pass through  $M - 1$  processes before we get back to 1.

Using the notation for cyclic distance, we can characterize the situation that  $t$  is strictly between  $n$  and  $k$  by:

$$0 < \delta(t, k) < \delta(t, n).$$

This inequality states that  $t$  is different from both  $k$  and  $n$  and, as  $t$  keeps progressing in cyclic order, it will reach  $k$  before it reaches  $n$ . That is,  $t$  is closer to  $k$  than to  $n$ . This situation is illustrated in Fig. 3.16.

- $t = k$ , but the resource has not yet been granted to  $C[k]$

This can be described by

$$t = k \wedge \text{at\_}m_{0,1}.$$

Combining the two cases, and adding the fact that  $C[k]$  is waiting at  $\ell_2$ , we obtain

$$\varphi_1: \text{at\_}\ell_2[k] \wedge \left( 0 < \delta(t, k) < \delta(t, n) \vee (t = k \wedge \text{at\_}m_{0,1}) \right).$$

- *Constructing  $\varphi_2$*

Next, we consider  $\varphi_2$ , which should characterize states in which  $C[n]$  owns the resource and  $C[k]$  is still waiting at  $\ell_2$ . These can be described by

$$\varphi_2: \text{at\_}\ell_2[k] \wedge t = n \wedge \text{at\_}m_2.$$

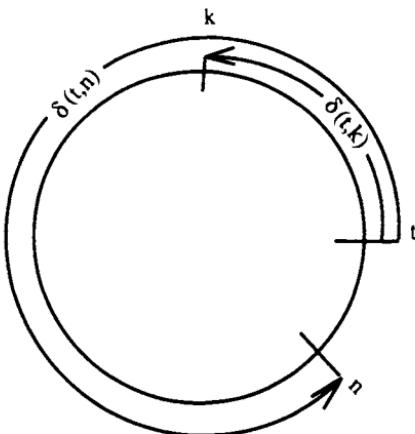


Fig. 3.16.  $k$  is closer to  $t$  than  $n$  in cyclic scanning order.

- *Constructing  $\varphi_3$*

Finally, we consider  $\varphi_3$ , which should characterize states in which  $C[n]$  may own the resource ahead of  $C[k]$ . Such states may arise after the last interaction with  $C[k]$  and before the next interaction with  $C[n]$ . As in the case of  $\varphi_1$ , we have to consider two subcases.

- $t$  is after  $k$  and before  $n$

This can be described by

$$0 < \delta(t, n) < \delta(t, k),$$

which claims that  $t$  is closer in cyclic order to  $n$  than it is to  $k$ .

- $t = n$ , but the resource has not yet been granted to  $C[n]$ .

This can be described by:

$$t = n \wedge at\_m_{0,1}.$$

Combining these two cases, and adding the fact that  $C[k]$  is waiting at  $\ell_2$ , we obtain:

$$\varphi_3: at\_\ell_2[k] \wedge \left( 0 < \delta(t, n) < \delta(t, k) \vee (t = n \wedge at\_m_{0,1}) \right).$$

### Applying Rule NWAIT

With these definitions of  $\varphi_0, \dots, \varphi_3$ , we check the premises of rule NWAIT.

- Show N1:  $at\_\ell_2[k] \rightarrow (\varphi_0 \vee \dots \vee \varphi_3)$

We have to show that all the accessible states satisfying  $at\_\ell_2[k]$  must satisfy one of  $\varphi_0 - \varphi_3$ . This is done by case analysis.

*Case:*  $t = n$ .

Assertion  $\varphi_2$  covers all the states satisfying

$$at\_l_2[k] \wedge t = n \wedge at\_m_2.$$

Assertion  $\varphi_3$  covers all the states satisfying

$$at\_l_2[k] \wedge t = n \wedge at\_m_{0,1}.$$

Since  $at\_m_{0..2}$  is an invariant of the program, it follows that

$$at\_l_2[k] \wedge t = n \rightarrow \varphi_2 \vee \varphi_3.$$

*Case:*  $t = k$ .

Assertion  $\varphi_1$  covers all the states satisfying

$$at\_l_2[k] \wedge t = k \wedge at\_m_{0,1}.$$

Due to the invariant  $\chi[k]$ :  $at\_l_{3,4}[k] \leftrightarrow (at\_m_2 \wedge t = k)$ ,  $at\_l_2[k] \wedge t = k$  implies  $\neg at\_m_2$ , from which we get

$$at\_l_2[k] \wedge t = k \rightarrow \varphi_1.$$

*Case:*  $t \neq k, t \neq n$ .

Assertion  $\varphi_1$  covers all the states satisfying

$$at\_l_2[k] \wedge 0 < \delta(t, k) < \delta(t, n).$$

Assertion  $\varphi_3$  covers all the states satisfying

$$at\_l_2[k] \wedge 0 < \delta(t, n) < \delta(t, k).$$

Since  $k \neq n$ , it follows that  $\delta(t, k) \neq \delta(t, n)$ , and both of them are positive, which shows that one of the two cases must hold. It follows that

$$at\_l_2[k] \wedge t \neq k \wedge t \neq n \rightarrow \varphi_1 \vee \varphi_3.$$

- Show N2:  $\varphi_i \rightarrow q_i$  for  $i = 0, \dots, 3$

By the invariant

$$\chi[i]: at\_l_{3,4}[i] \leftrightarrow at\_m_2 \wedge t = i,$$

- Assertion  $\varphi_0$ :  $t = k \wedge at\_m_2$  implies  $q_0$ :  $at\_l_{3,4}[k]$ .
- Assertion  $\varphi_1$  implies  $t \neq n$  (since  $k \neq n$ ) which, by  $\chi[n]$ , implies  $q_1$ :  $\neg at\_l_{3,4}[n]$ .
- Assertion  $\varphi_2$  implies  $t = n \wedge at\_m_2$  which, by  $\chi[n]$ , implies  $q_2$ :  $at\_l_{3,4}[n]$ .
- Assertion  $\varphi_3$  implies that either  $t \neq n$  or  $at\_m_{0,1}$  holds. By  $\chi[n]$ , this implies  $q_3$ :  $\neg at\_l_{3,4}[n]$ .

- Show N3:  $\{\varphi_i\} T \left\{ \bigvee_{j \leq i} \varphi_j \right\}$  for  $i = 1, 2, 3$

We present the relations between these assertions in the WAIT diagram of Fig. 3.17.

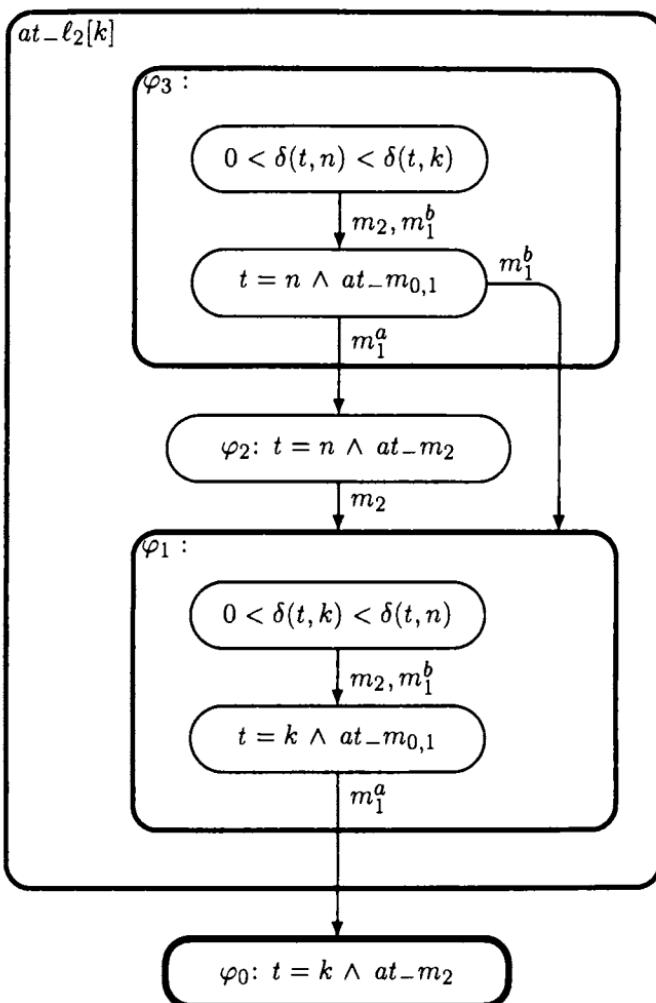


Fig. 3.17. WAIT diagram.

This concludes the proof of  $\psi$ .

In **Problem 3.12** and **Problem 3.13**, we ask the reader to perform a similar overtaking analysis for shared-variables versions of the resource-allocator program. **Problem 3.14** requests a similar analysis of Program MUX-SWAP.

## \* 3.5 Completeness

Rule NWAIT was proposed as a general rule for establishing nested waiting-for properties of a fair transition system  $P$ . We will show that the rule is *complete*. As discussed in Section 2.5 (completeness of rule INV), we actually show completeness relative to first-order reasoning. The precise statement of completeness is

**Theorem 3.3** (completeness of rule NWAIT)

If the formula

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0$$

is  $P$ -valid, there exist assertions  $\varphi_0, \varphi_1, \dots, \varphi_m$  such that the premises of rule NWAIT are provable from state validities.

In the following, we assume a given transition system  $P = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$  and refer to  $P$ -accessible states simply as accessible. We introduce first the notions of segments satisfying formulas and ranks of states.

### Segments Satisfying Waiting-for Formulas

Let  $\sigma: s_0, s_1, \dots$  be a  $P$ -computation. A ( $P$ )-segment is a finite sequence of states

$$[s_a, \dots, s_b],$$

such that  $b \geq a$  and for every  $i$ ,  $a \leq i < b$ ,  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$  for some  $\tau \in \mathcal{T}$ . We say that the segment  $[s_a, \dots, s_b]$  originates in  $s_a$ .

We will denote the segment  $[s_a, \dots, s_b]$  by  $\sigma[a..b]$  and the segment  $[s_a, \dots, s_{b-1}]$  by  $\sigma[a..b)$ . The infinite segment  $s_a, \dots$  is In the following, we focus on a nested waiting-for formula

$$\psi_m: q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0,$$

and also consider subformulas of  $\psi_m$  of the form

$$\psi_j: q_j \mathcal{W} q_{j-1} \cdots q_1 \mathcal{W} q_0,$$

for  $j = 0, \dots, m$ .

Recall that a computation  $\sigma$  satisfies  $q_j \mathcal{W} q_{j-1} \cdots q_1 \mathcal{W} q_0$  at position  $a \geq 0$  if there exist indices (possibly  $\infty$ )

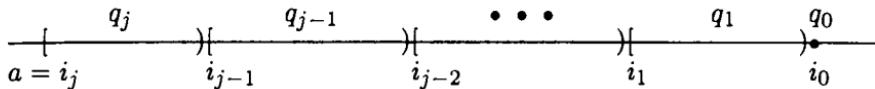
$$a = i_j \leq \cdots \leq i_1 \leq i_0 \leq \infty$$

such that

$\sigma[i_j..i_{j-1}]$  is a  $q_j$ -interval,  
 $\sigma[i_{j-1}..i_{j-2}]$  is a  $q_{j-1}$ -interval,  
 $\vdots$   
 $\sigma[i_1..i_0]$  is a  $q_1$ -interval,  
and if  $i_0 < \infty$  then  $q_0$  is satisfied by  $s_{i_0}$ .

This definition allows the case  $i_{n-1} = \dots = i_0 = \infty$  for some  $n \leq j$ , and then the requirement for  $q_n$  is that it is satisfied by  $s_t$ , for all  $t \geq i_n$ . In that case, the requirements for  $q_{n-1}, \dots, q_0$  are satisfied vacuously. The definition also allows the case of equality  $i_k = i_{k-1}$ , for some  $k \in [1..j]$ . In this case, the interval  $\sigma[i_k..i_{k-1}]$  is empty, and  $q_k$  holds over an empty interval.

This definition can be illustrated by the figure



Inspired by this definition, we say that the segment  $\sigma[a..b]$  satisfies the nested formula  $q_j \mathcal{W} q_{j-1} \dots q_1 \mathcal{W} q_0$  if there exist indices

$$a = i_j \leq \dots \leq i_1 \leq i_0 \leq b + 1,$$

such that

- for every  $r = 1, \dots, j$ ,  $\sigma[i_r..i_{r-1}]$  is a  $q_r$ -interval, and
- if  $i_0 < b + 1$  then  $q_0$  is satisfied by  $s_{i_0}$ .

As before, this definition allows the case  $i_{n-1} = \dots = i_0 = b + 1$  for some  $n \leq j$ , and then the last nonvacuous requirement is that  $\sigma[i_n..b]$  is a  $q_n$ -interval.

For the special case that  $a = b$ , i.e., a singleton segment consisting of the state  $s_a$ ,  $[s_a]$  satisfies  $\psi_j$  iff  $s_a \models q_k$  for some  $k \in [0..j]$ .

The similarity between the definitions of satisfaction for infinite and finite sequences leads to the following property:

**Claim 3.4** (segment satisfaction)

If the formula

$$\psi_j: q_j \mathcal{W} q_{j-1} \dots q_1 \mathcal{W} q_0$$

holds at position  $a \geq 0$  of a computation  $\sigma: s_0, s_1, s_2, \dots$  then, for every  $b \geq a$ , the segment  $\sigma[a..b]$  satisfies  $\psi_j$ .

**Justification** Let  $a = i_j \leq \dots \leq i_0 \leq \infty$  be the indices supporting the claim that  $\psi_j$  holds at  $a$ . For each  $r = 0, \dots, j$ , define  $\hat{i}_r = \min(i_r, b + 1)$ . It is not

difficult to see that the indices  $a = \hat{i}_j \leq \dots \leq \hat{i}_0 \leq b + 1$  serve to establish the satisfaction of  $\psi_j$  by the segment  $\sigma[a..b]$ . ■

**Example** Consider a transition system **SYS-A** with the components:

*System variables:*  $\{y: \text{integer}\}$ .

*Initial condition:*  $\Theta: y = 0$ .

*Transitions:*  $T: \{\tau_I, \tau_5, \tau_r\}$  with transition relations:

$$\rho_I: y' = y \quad \rho_5: y' = y + 5 \quad \rho_r: y' = 0.$$

The computation

$$\begin{aligned} \sigma: \quad s_0: & \langle y: 0 \rangle, \quad s_1: \langle y: 5 \rangle, \quad s_2: \langle y: 10 \rangle, \quad s_3: \langle y: 15 \rangle, \quad s_4: \langle y: 20 \rangle, \\ & s_5: \langle y: 20 \rangle, \quad \dots, \quad s_i: \langle y: 20 \rangle, \end{aligned}$$

satisfies

$$\psi_2: \underbrace{y \geq 10}_{q_2} \mathcal{W} \underbrace{y \geq 20}_{q_1} \mathcal{W} \underbrace{y = 0}_{q_0}$$

at position 2, since  $\sigma[2..4)$  is a  $q_2$ -interval and  $\sigma[4..\infty)$  is a  $q_1$ -interval. The segment

$$[s_2: \langle y: 10 \rangle, \quad s_3: \langle y: 15 \rangle, \quad s_4: \langle y: 20 \rangle],$$

also satisfies  $\psi_2$  since  $\sigma[2..3)$  is a  $q_2$ -interval and  $\sigma[4..5)$  is a  $q_1$ -interval.

Even the shorter segment

$$[s_2: \langle y: 10 \rangle, \quad s_3: \langle y: 15 \rangle]$$

satisfies  $\psi_2$ , since  $\sigma[2..4)$  is a  $q_2$ -interval. ■

## Ranks of States

For a nested formula  $q_m \mathcal{W} \dots \mathcal{W} q_0$ , we say that a state  $s$  has rank  $j$ ,  $0 \leq j \leq m$ , if all segments originating at  $s$  satisfy

$$\psi_j: q_j \mathcal{W} q_{j-1} \dots q_1 \mathcal{W} q_0.$$

If  $j$  is the smallest nonnegative integer such that  $s$  has rank  $j$  then  $s$  is said to have the minimal rank  $j$ .

**Example** Consider a transition system **SYS-B** with the following components:

*System variables:*  $\{y: \text{integer}\}$ .

*Initial condition:*  $\Theta: y < 0$ .

*Transitions:*  $T: \{\tau_I, \tau_<, \tau_2, \tau_{11}\}$  with the following transition relations

$$\rho_1: y' = y$$

$$\rho_2: y' = y - 2$$

$$\rho_{<}: y < 0 \wedge y' \leq 20$$

$$\rho_{11}: y' = y - 11.$$

Note that  $\tau_{<}$  is enabled only when  $y$  is negative and, when taken, it assigns to  $y$  an arbitrary value not exceeding 20.

Consider the nested waiting-for formula

$$\underbrace{\text{even}(y)}_{q_3} \mathcal{W} \underbrace{\text{odd}(y)}_{q_2} \mathcal{W} \underbrace{\text{even}(y)}_{q_1} \mathcal{W} \underbrace{y < 0}_{q_0}.$$

Let us classify the states of sys-B (not necessarily the accessible ones) according to their minimal ranks:

- All states with  $y < 0$  have the minimal rank 0. This is because they satisfy  $q_0$ .
- All states that satisfy  $0 \leq y \leq 10 \wedge \text{even}(y)$  have a minimal rank 1. For example, the state  $\langle y: 10 \rangle$  initiates the segment

$$\left[ \underbrace{\langle y: 10 \rangle}_{\models \text{even}(y)}, \underbrace{\langle y: -1 \rangle}_{\models y < 0} \right],$$

which satisfies  $\text{even}(y) \mathcal{W} (y < 0)$  but not the shorter formula  $y < 0$ .

- All states that satisfy  $1 \leq y \leq 21 \wedge \text{odd}(y)$  have the minimal rank 2. For example, the state  $\langle y: 21 \rangle$  initiates the segment

$$\left[ \underbrace{\langle y: 21 \rangle}_{\models \text{odd}(y)}, \underbrace{\langle y: 10 \rangle, \langle y: 8 \rangle}_{\models \text{even}(y)}, \underbrace{\langle y: -3 \rangle}_{\models y < 0} \right],$$

which satisfies  $\text{odd}(y) \mathcal{W} \text{even}(y) \mathcal{W} (y < 0)$  but not the shorter formula  $\text{even}(y) \mathcal{W} (y < 0)$ .

- All states that satisfy  $12 \leq y \leq 32 \wedge \text{even}(y)$  have the minimal rank 3. For example, the state  $\langle y: 32 \rangle$  initiates the segment

$$\left[ \underbrace{\langle y: 32 \rangle}_{\models \text{even}(y)}, \underbrace{\langle y: 21 \rangle}_{\models \text{odd}(y)}, \underbrace{\langle y: 10 \rangle}_{\models \text{even}(y)}, \underbrace{\langle y: -1 \rangle}_{\models y < 0} \right],$$

which satisfies  $\text{even}(y) \mathcal{W} \text{odd}(y) \mathcal{W} \text{even}(y) \mathcal{W} (y < 0)$ .

- All other states do not have a defined rank. For example,  $\langle y: 33 \rangle$  initiates the segment

$$[\langle y: 33 \rangle, \langle y: 22 \rangle, \langle y: 11 \rangle, \langle y: 0 \rangle],$$

which does not satisfy  $\text{even}(y) \mathcal{W} \text{odd}(y) \mathcal{W} \text{even}(y) \mathcal{W} (y < 0)$ . ■

There are several properties that follow from the preceding definitions.

**Property P1** If  $s_a, \dots, s_b$ ,  $a < b$ , is a segment satisfying the formula  $\psi_j: q_j \mathcal{W} q_{j-1} \cdots q_1 \mathcal{W} q_0$  and  $s_a$  does not satisfy  $q_0$ , then the shorter segment  $s_{a+1}, \dots, s_b$  also satisfies  $q_j \mathcal{W} q_{j-1} \cdots q_1 \mathcal{W} q_0$ .

This is a direct consequence of the definition of satisfaction by segments, and the observation that if  $[s_{a+1}, \dots, s_b]$  satisfies  $\psi_n: q_n \mathcal{W} q_{n-1} \cdots q_1 \mathcal{W} q_0$  for some  $n < j$  then it also satisfies  $\psi_j$  by taking  $i_n = \dots = i_j$  to be  $a + 1$ .

**Property P2** If a state  $s$  has a minimal rank  $j > 0$  then any  $s'$ , a successor of  $s$ , has a minimal rank  $i \leq j$ .

Let  $s$  have the minimal rank  $j > 0$  and  $s'$  be a successor of  $s$ . We will show that  $s'$  also has rank  $j$ . Let  $[s' = s_a, \dots, s_b]$  be a segment originating at  $s'$ . Then  $[s, s_a, \dots, s_b]$  is a segment originating at  $s$ . Since  $s$  has rank  $j > 0$ , the segment  $[s, s_a, \dots, s_b]$  satisfies  $\psi_j$ . As the minimal rank of  $s$  is positive,  $s$  does not satisfy  $q_0$ . By Property P1, also the tail segment  $[s_a, \dots, s_b]$  satisfies  $\psi_j$ . This shows that every segment originating at  $s'$  satisfies  $\psi_j$ , establishing that  $s'$  has the rank  $j$ . Consequently, the minimal rank of  $s'$  cannot exceed  $j$ .

### Assertions Characterizing Minimal Ranks

The completeness proof proceeds by constructing assertions  $\varphi_0, \varphi_1, \dots, \varphi_m$  such that

$$(*) \quad s \models \varphi_j \text{ iff } s \text{ has the minimal rank } j.$$

We will defer the actual construction to a later subsection and proceed under the assumption that such assertions have been constructed. The construction will take  $\varphi_0$  to be  $q_0$ .

**Example** For the transition system SYS-B introduced earlier and the waiting-for formula

$$\underbrace{\text{even}(y)}_{q_3} \mathcal{W} \underbrace{\text{odd}(y)}_{q_2} \mathcal{W} \underbrace{\text{even}(y)}_{q_1} \mathcal{W} \underbrace{y < 0}_{q_0}$$

we have identified the following characterizing assertions

$$\begin{array}{ll} \varphi_0: & y < 0 \\ \varphi_1: & 0 \leq y \leq 10 \wedge \text{even}(y) \\ \varphi_2: & 1 \leq y \leq 21 \wedge \text{odd}(y) \\ \varphi_3: & 12 \leq y \leq 32 \wedge \text{even}(y). \end{array}$$

■

Assume that the formula

$$p \Rightarrow \underbrace{q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0}_{\psi_m}$$

is  $P$ -valid. Recall that  $\psi_m$  specifies the initiation of a succession of intervals, starting with a  $q_m$ -interval and, either terminating at a  $q_0$ -position, or extending to infinity.

We will show that, taking  $\varphi_0, \varphi_1, \dots, \varphi_m$  to be the assertions characterizing minimal ranks as defined in (\*), all premises of rule NWAIT are  $P$ -state valid.

### Premise N1

Consider an accessible state  $s$  that satisfies  $p$ . We will show that  $s$  also satisfies one of  $\varphi_0, \varphi_1, \dots, \varphi_m$ . That is, state  $s$  has the rank  $m$ .

To see this, let

$$[s = s_a, \dots, s_b]$$

be any segment originating at  $s$ . Since  $s$  is accessible, this segment can be embedded in a computation

$$\sigma: s_0, \dots, s_a, \dots, s_b, \dots$$

In particular,  $s_a = s$  satisfies  $p$ . Since  $p \Rightarrow \psi_m$  is  $P$ -valid,  $\psi_m$  holds at position  $a$  in  $\sigma$ . By the segment-satisfaction claim, the segment  $[\hat{s}_1, \dots, \hat{s}_k]$  satisfies  $\psi_m$ . As this conclusion holds for each segment originating at  $s$ , it follows that  $s$  has the rank  $m$ . Therefore, it has some minimal rank  $r \in [0..m]$ . As assertion  $\varphi_r$  characterizes all states having  $r$  as their minimal rank, state  $s$  satisfies  $\varphi_r$  and therefore satisfies the disjunction  $\bigvee_{j=0}^m \varphi_j$ .

This shows that every accessible state that satisfies  $p$  also satisfies  $\bigvee_{j=0}^m \varphi_j$ , i.e., that the implication

$$p \rightarrow \bigvee_{j=0}^m \varphi_j$$

is  $P$ -state valid. By the completeness theorem of rule INV, if an assertion  $\psi$  is  $P$ -state valid (hence  $P$ -invariant), this can be proven from state validities. We conclude that premise N1 can be proven from state validities.

### Premise N2

Let  $s$  be a state satisfying  $\varphi_i$ , for some  $i = 0, \dots, m$ . We will show that it also satisfies  $q_i$ .

If  $i = 0$  then  $s$  satisfies  $\varphi_0$  which is equivalent to  $q_0$ .

Next, consider the case of  $i > 0$ . Assume, to the contrary, that  $q_i$  is false on  $s$ . Consider an arbitrary segment  $[s = s_a, \dots, s_b]$  originating at  $s$ . By the definition of  $\varphi_i$  in (\*),  $\psi_i$  is satisfied by  $[s_a, \dots, s_b]$ . However, since  $q_i$  is false on  $s_a$ , this segment must also satisfy  $\psi_{i-1}$ . This shows that  $s$  also has the rank  $i - 1$ , contradicting the fact that  $i$  is the minimal rank of  $s$ .

This proves that every state satisfying  $\varphi_i$  must also satisfy  $q_i$ , i.e.,

$$\varphi_i \rightarrow q_i$$

is state valid (and therefore also  $P$ -state valid).

### Premise N3

Let  $s$  be a state satisfying  $\varphi_i$ , for some  $i = 1, \dots, m$ , and let  $s'$  be a successor of  $s$ . We will show that  $s'$  satisfies  $\bigvee_{j \leq i} \varphi_j$ .

By the definition of  $\varphi_i$ , state  $s$  has minimal rank  $i$ . By property P2,  $s'$  has a minimal rank  $j$ ,  $0 \leq j \leq i$ . It follows that  $s'$  satisfies  $\varphi_j$  and therefore satisfies  $\bigvee_{j \leq i} \varphi_j$ .

This shows that the implication

$$\rho_T \wedge \varphi_i \rightarrow \bigvee_{j \leq i} \varphi'_j$$

is state valid.

This concludes the proof of completeness of rule NWAIT.

**Example** Let us return to the transition system **sys-B** discussed in the previous examples. We consider the formula

$$y \geq 0 \Rightarrow \underbrace{\text{even}(y)}_{q_3} \mathcal{W} \underbrace{\text{odd}(y)}_{q_2} \mathcal{W} \underbrace{\text{even}(y)}_{q_1} \mathcal{W} \underbrace{y < 0}_{q_0},$$

and will prove it to be valid over system **sys-B**, using rule NWAIT and the assertions

$$\begin{array}{ll} \varphi_0: y < 0 & \varphi_1: 0 \leq y \leq 10 \wedge \text{even}(y) \\ \varphi_2: 1 \leq y \leq 21 \wedge \text{odd}(y) & \varphi_3: 12 \leq y \leq 32 \wedge \text{even}(y) \end{array}$$

which characterize the states having minimal ranks  $0, \dots, 3$ , respectively.

- Premise N1 requires showing the  $P$ -state validity of

$$y \geq 0 \rightarrow \underbrace{y < 0}_{\varphi_0} \vee \underbrace{0 \leq y \leq 32 \wedge \text{even}(y)}_{\varphi_1 \vee \varphi_3} \vee \underbrace{1 \leq y \leq 21 \wedge \text{odd}(y)}_{\varphi_2}.$$

Using rule INV, we can prove the invariance over **sys-B** of

$$y \leq 20.$$

It is easy to show that  $y \leq 20$  establishes N1 as state valid over **sys-B**.

- Premise N2 requires showing  $\varphi_i \rightarrow q_i$  for  $i = 0, \dots, 3$ . All these implications are state valid.

- Premise N3 requires showing

$$\rho_\tau \wedge \varphi_i \rightarrow \varphi'_0 \vee \cdots \vee \varphi'_i,$$

for  $i = 1, 2, 3$  and transition  $\tau$ . We will consider each transition separately.

- Transition  $\tau_<$  is disabled on all  $\varphi_i$ -states,  $i > 0$ , so the implication is trivially state valid.
- Transition  $\tau_2$  does not change the parity of  $y$  and decreases its value by 2. Consequently,

$$y' = y - 2 \wedge \underbrace{0 \leq y \leq 10 \wedge \text{even}(y)}_{\varphi_1} \rightarrow \underbrace{y' < 0}_{\varphi'_0} \vee \underbrace{0 \leq y' \leq 10 \wedge \text{even}(y')}_{\varphi'_1}$$

$$y' = y - 2 \wedge \underbrace{1 \leq y \leq 21 \wedge \text{odd}(y)}_{\varphi_2} \rightarrow \underbrace{y' < 0}_{\varphi'_0} \vee \underbrace{1 \leq y' \leq 21 \wedge \text{odd}(y')}_{\varphi'_2}$$

$$y' = y - 2 \wedge \underbrace{12 \leq y \leq 32 \wedge \text{even}(y)}_{\varphi_3} \rightarrow \underbrace{y' < 0}_{\varphi'_0} \vee \underbrace{0 \leq y' \leq 10 \wedge \text{even}(y')}_{\varphi'_1} \vee \underbrace{12 \leq y' \leq 32 \wedge \text{even}(y')}_{\varphi'_3}.$$

- Transition  $\tau_{11}$  — we claim the following implications

$$y' = y - 11 \wedge \underbrace{0 \leq y \leq 10 \wedge \text{even}(y)}_{\varphi_1} \rightarrow \underbrace{y' < 0}_{\varphi'_0}$$

$$y' = y - 11 \wedge \underbrace{1 \leq y \leq 21 \wedge \text{odd}(y)}_{\varphi_2} \rightarrow \underbrace{y' < 0}_{\varphi'_0} \vee \underbrace{0 \leq y' \leq 10 \wedge \text{even}(y')}_{\varphi'_1}$$

$$y' = y - 11 \wedge \underbrace{12 \leq y \leq 32 \wedge \text{even}(y)}_{\varphi_3} \rightarrow \underbrace{1 \leq y' \leq 21 \wedge \text{odd}(y')}_{\varphi'_2},$$

which are state validities. ■

### Constructing the Assertions $\varphi_j$

The last missing detail in the completeness proof is the construction of the assertions  $\varphi_j$ ,  $j = 0, \dots, m$ , characterizing the sets of states with minimal rank  $j$ .

Assume that  $P$  is a transition system with system variables  $y_1, \dots, y_k$ , ranging over the data domain  $D$ .

For  $j = 0$ , we define  $\varphi_0 = q_0$ .

Consider the case  $j > 0$ . Let  $a \in [1..n] \times [1..k] \mapsto D$  be a two-dimensional array over the domain  $D$ . The assertion  $\text{index}_j(a, n)$  claims that a segment  $s_1, \dots, s_n$ , encoded by the array  $A$ , has a set of indices necessary for establishing that  $s_1, \dots, s_n$  satisfies  $q_j \mathcal{W} q_{j-1} \dots q_1 \mathcal{W} q_0$ . It is given by

$$\text{index}_j(a, n): \exists i_j, \dots, i_1, i_0 \left( \begin{array}{l} 1 = i_j \leq \dots \leq i_1 \leq i_0 \leq n + 1 \\ \wedge \\ \bigwedge_{r=1}^j \forall t (i_r \leq t < i_{r-1}): q_r(a[t]) \\ \wedge \\ i_0 \leq n \rightarrow q_0(a[i_0]) \end{array} \right).$$

As in Section 2.5, we use the notation  $a[t]$ , for  $t \in [1..n]$ , to denote the  $t$ -th row of the array, given by  $a[t, 1], \dots, a[t, k]$ . Such a row encodes the state  $s_t$  by recording the values of the system variables  $\bar{y} = (y_1, \dots, y_k)$  in the entries  $a[t, 1] = s_t[y_1], \dots, a[t, k] = s_t[y_k]$ . The subformula  $q_r(a[t])$  appearing in the formula  $\text{index}_j(a, n)$  is the truth-value of assertion  $q_r$  evaluated by interpreting each  $y_i$  as  $a[t, i]$ . In a similar way,  $q_0(a[i_0])$  is  $q_0$  evaluated with the interpretation  $\bar{y} = a[i_0]$ .

Thus, formula  $\text{index}_j(a, n)$  states that there exists a sequence of indices  $1 = i_j \leq \dots \leq i_0 \leq n + 1$ , such that  $q_r$  holds on the states encoded in the subarray  $a[i_r..i_{r-1}]$  for each  $r \in [1..j]$  and, if  $i_0 \leq n$ , then  $q_0$  holds on the state encoded by  $a[i_0]$ .

Assertion  $\text{has\_rank}_j(\bar{y})$  claims that the state represented by  $\bar{y}$  has rank  $j$ :

$$\text{has\_rank}_j(\bar{y}): \forall n > 0 \forall a \in [1..n] \times [1..k] \mapsto D : \bar{y} = a[1] \wedge \text{evolve}(a, n) \rightarrow \text{index}_j(a, n),$$

where

$$\text{evolve}(a, n): \forall t (1 \leq t < n): \bigvee_{\tau \in T} \rho_\tau(a[t], a[t+1]).$$

Assertion  $\text{has\_rank}_j(\bar{y})$  claims that, for every array  $a$  of dimensions  $n \times k$  encoding a segment originating at  $\bar{y}$ , the assertion  $\text{index}_j(a, n)$  is true.

The assertion  $\varphi_j$  can now be defined as

$$\varphi_j(\bar{y}): \text{has\_rank}_j(\bar{y}) \wedge \bigwedge_{i=0}^{j-1} \neg \text{has\_rank}_i(\bar{y}).$$

This formula states that  $\bar{y}$  has the rank  $j$  but no smaller rank.

**Problem 3.15** requests that the reader present an alternative completeness proof, based on a different principle.

## \* 3.6 Finite-State Algorithmic Verification

We now consider the special case of finite-state programs and present an algorithm for checking that a finite-state program  $P$  satisfies a given nested waiting-for formula.

Assume that the formula to be checked is

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0.$$

For convenience, we extend the sequence of assertions and define

$$q_{m+1}: \top.$$

For each  $j \in [0..m+1]$ , let  $\psi_j$  be the subformula

$$\psi_j: q_j \mathcal{W} q_{j-1} \cdots q_1 \mathcal{W} q_0.$$

In particular,

$$\psi_{m+1}: \top \mathcal{W} q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0,$$

which means that  $\psi_{m+1}$  is equivalent to  $\top$ .

As a first step we construct the state-transition graph  $G_P$  which contains all the  $P$ -accessible states, and has an edge connecting  $s$  to  $\tilde{s}$  iff  $\tilde{s}$  is a successor of  $s$ .

As in page 290, we say that  $s$  has rank  $j$  if all segments (finite paths in  $G_P$ ) departing from  $s$  satisfy  $\psi_j$ . We observe the following properties of states and their ranks:

- Every state  $s$  has the rank  $m+1$ . This is because every segment departing from  $s$  can be viewed as a single  $q_{m+1}$ -interval since  $q_{m+1} = \top$ .
- If state  $s$  has the rank  $j \in [1..m+1]$  it also has the rank  $k$ , for every  $k \in [j..m+1]$ .

The main part of the algorithm determines for each state  $s$  its minimal rank, which we denote by  $\mathcal{R}(s)$ . Since every state has at least the rank  $m+1$ ,  $\mathcal{R}(s)$  is defined for any state and is not higher than  $m+1$ .

The question is how to determine  $\mathcal{R}(s)$  in an efficient way, without exploring all paths in  $G_P$ . There are several properties that the minimal ranking  $\mathcal{R}$  satisfies.

R1. If  $s \Vdash q_0$  then  $\mathcal{R}(s) = 0$ .

R2. If  $\mathcal{R}(s) = j$  then  $j \in [0..m+1]$  and  $s \Vdash q_j$ .

R3. If  $\mathcal{R}(s) > 0$  and  $s'$  is a successor of  $s$  then  $\mathcal{R}(s) \geq \mathcal{R}(s')$ .

Properties R1 and R2 follow directly from the definitions of ranks and minimal ranks. Property R3 is identical to property P2, stated and proven in page 292.

The notions of ranks and minimal ranks relate directly to *segments* departing from a state. Thus, a state  $s$  has a rank  $j \leq m+1$  if and only if all segments (i.e.

finite paths) that depart from  $s$  satisfy  $\psi_j$ . However, for checking validity over a finite-state system, we need to establish a connection between ranks and infinite paths departing from a state  $s$  in  $G_P$ . This is done in the following claim:

**Claim 3.5 (ranks and infinite paths)**

A state  $s$  in  $G_P$  has a rank  $j \leq m + 1$  iff all infinite paths departing from  $s$  satisfy  $\psi_j$ .

**Justification** Since the case  $j = m + 1$  is trivial ( $\psi_{m+1}$  is equivalent to T), we restrict our attention to  $j \leq m$ . In one direction, assume that all infinite paths departing from a state  $s$  satisfy  $\psi_j$ . By Claim 3.4, if an infinite path  $\sigma$  satisfies  $\psi_j$ , then so do all of its finite prefixes, i.e., all segments that are prefixes of  $\sigma$ . It follows that all segments departing from  $s$  satisfy  $\psi_j$ , implying that  $s$  has the rank  $j$ .

In the other direction, let  $\sigma: s_a, s_{a+1}, \dots$  be an infinite path departing from  $s$ , i.e.,  $s_a = s$ . We will show that  $\sigma$  satisfies  $\psi_j$ . Let  $r_a, r_{a+1}, \dots$  be the sequence of minimal ranks of the states in  $\sigma$ . There are two cases to consider:

**Case I:** The rank sequence contains a finite prefix of the form:

$$r_a \geq r_{a+1} \geq \dots \geq r_{b-1} > r_b = 0$$

such that  $a \leq b$  and  $r_i > 0$  for all  $i$ ,  $a \leq i < b$ . By property R2,  $s_j \Vdash q_j$ , for each  $j$ ,  $a \leq j \leq b$ . It follows that  $\sigma$  satisfies  $\psi_{r_a}$  and, since  $j \geq r_a$  ( $r_a$  being the minimal rank of  $s = s_a$ ),  $\sigma$  also satisfies  $\psi_j$ .

**Case II:**  $r_i > 0$  for all  $i \geq a$ . In this case, the rank sequence is infinite and monotonically nondecreasing:

$$r_a \geq r_{a+1} \geq \dots$$

Also in this case,  $\sigma$  satisfies  $\psi_{r_a}$  and, therefore,  $\sigma$  satisfies  $\psi_j$ . ■

## An Algorithm for Determination of Minimal Ranks

Before presenting the algorithm we remind the reader of some notions of graph theory.

Let  $G = (N, E)$  be a directed graph consisting of a set of nodes  $N$  and a set of edges  $E$  connecting the nodes. A subgraph  $G_1 = (N_1, E_1)$  is obtained by taking a subset  $N_1 \subseteq N$  of the nodes and all edges connecting nodes of  $N_1$ . A path in  $G$  is called an  $N_1$ -path if it passes only through nodes belonging to  $N_1$ . Since subgraphs of  $(N, E)$  are uniquely determined by their set of nodes, we will specify a subgraph  $(N_1, E_1)$  by referring to  $N_1$  alone.

A subgraph  $N_1$  is called a *strongly connected subgraph* (scs) if for every two distinct nodes  $a, b \in N_1$ , there exists an  $N_1$ -path leading from  $a$  to  $b$  and an

$N_1$ -path leading from  $b$  to  $a$ . Note, in particular, that any subgraph consisting of a single node is, by definition, an SCS.

An SCS  $N_2$  is called a *maximal strongly connected subgraph* (MSCS for short) if it is not properly contained in any larger SCS.

There exist several efficient algorithms for decomposing a graph into a sequence of MSCS's. Such an algorithm takes as input a graph  $G = (N, E)$  and produces as output a sequence of MSCS's  $N_1, \dots, N_d$ , such that  $N = N_1 \cup \dots \cup N_d$  and, whenever there is an edge (or a path) from a node in  $N_i$  to a node in  $N_j$ , then  $i \leq j$ . The sequence  $N_1, \dots, N_d$  is called a *sorted decomposition* of the graph  $G$ . We generically refer to such an algorithm by the name DECOMPOSE.

For a sorted decomposition  $N_1, \dots, N_d$ , we say that  $N_j$  is a *disjoint successor* of  $N_i$  if  $N_j \neq N_i$  and there is an edge connecting a node (state) in  $N_i$  to a node in  $N_j$ . Obviously, if  $N_j$  is a disjoint successor of  $N_i$  then  $i < j$ .

**Example** Consider the graph of Fig. 3.18, where self-edges have been omitted.

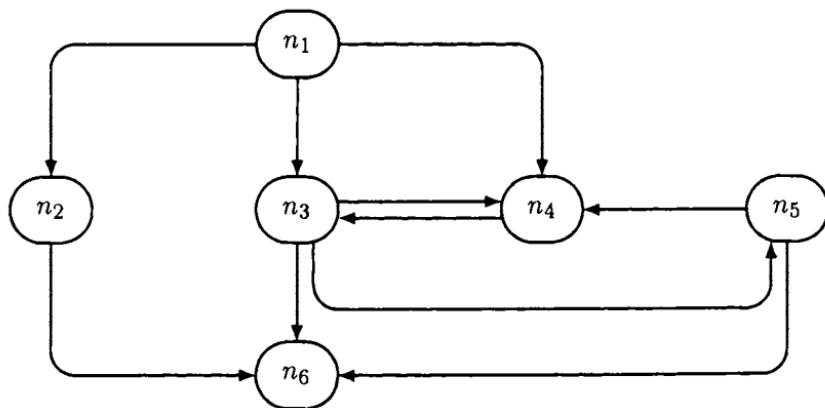


Fig. 3.18. Graph.

Obviously, each of the subgraphs consisting of a single node is strongly connected, and so are the subgraphs  $\{n_3, n_4\}$  and  $\{n_3, n_4, n_5\}$ . Note that  $\{n_4, n_5\}$  is not strongly connected even though there are paths from  $n_4$  to  $n_5$  and from  $n_5$  to  $n_4$ . The two possible sorted decompositions of this graph into MSCS's are:

$$\{n_1\}, \{n_2\}, \{n_3, n_4, n_5\}, \{n_6\} \quad \text{and} \quad \{n_1\}, \{n_3, n_4, n_5\}, \{n_2\}, \{n_6\}.$$

Among these subgraphs,  $\{n_2\}$  and  $\{n_3, n_4, n_5\}$  are disjoint successors of  $\{n_1\}$ , and  $\{n_6\}$  is a disjoint successor of both  $\{n_2\}$  and  $\{n_3, n_4, n_5\}$ . ■

For a set of states  $N$  and integer  $t \in [0..m + 1]$ , we define

$\text{lowest}(N, t)$ 

to be the smallest  $j \in [t..m+1]$ , such that  $s \Vdash q_j$  for all  $s \in N$ . As all states satisfy  $q_{m+1} = T$ ,  $\text{lowest}(N, t)$  is always defined, and will equal  $m+1$  in the worst case.

The following algorithm determines the minimal rank of all states in a finite-state system  $P$ :

**Algorithm MIN-RANK — minimal rank determination**

- *Step 1:* Construct the state-transition graph  $G_P$  for system  $P$ .
- *Step 2:* Construct the graph  $G_P^-$  by omitting from  $G_P$  all edges departing from  $q_0$ -states.
- *Step 3:* Obtain a sorted decomposition of  $G_P^-$  by Algorithm DECOMPOSE. Assume it is given by the MSCS sequence  $N_1, \dots, N_d$ .
- *Step 4:* Proceeding in reverse order for  $i = d, d-1, \dots, 1$ , compute for each  $N_i$  a minimal rank  $r_i$  that is expected to be the minimal rank of all states belonging to  $N_i$ .
  - If  $N_i$  has no disjoint successors then  $r_i = \text{lowest}(N_i, 0)$ .
  - If  $N_i$  has disjoint successors  $N_{j_1}, \dots, N_{j_b}$  with corresponding ranks  $r_{j_1}, \dots, r_{j_b}$ , let  $m_i$  be the maximal value among  $\{r_{j_1}, \dots, r_{j_b}\}$ . Define  $r_i = \text{lowest}(N_i, m_i)$ .

**Example** Consider the state-transition graph of Fig. 3.19.

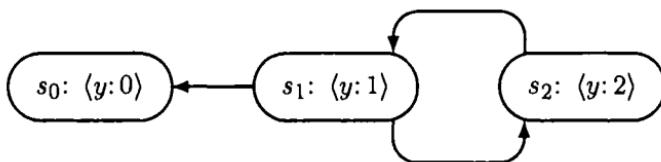


Fig. 3.19. State-transition graph.

Let us consider the formula

$$y \leq 1 \Rightarrow q_{19} \mathcal{W} q_{18} \cdots q_1 \mathcal{W} q_0,$$

where

$$q_0: y = 0$$

$$q_i: \text{odd}(y) \quad \text{for each } i = 1, 3, 5, \dots, 19,$$

$q_i: \text{even}(y) \quad \text{for each } i = 2, 4, \dots, 18.$

Thus, state  $s_0$  satisfies  $q_0, q_2, \dots, q_{18}$ , state  $s_1$  satisfies  $q_1, q_3, \dots, q_{19}$  and state  $s_2$  satisfies  $q_2, q_4, \dots, q_{18}$ .

Step 3 of algorithm MIN-RANK yields the sorted decomposition

$\{s_1, s_2\}, \{s_0\}.$

The rank of  $s_0$  is determined to be 0. To determine the rank of  $\{s_1, s_2\}$  we observe that  $\{s_0\}$  with rank 0 is the only disjoint successor  $\{s_1, s_2\}$  has. Consequently, we take the rank of  $\{s_1, s_2\}$  to be

$$\text{lowest}(\{s_1, s_2\}, 0) = 20$$

due to the fact that there is no  $j \in [0..19]$ , such that both  $s_1$  and  $s_2$  satisfy  $q_j$ . Therefore, since both  $s_1$  and  $s_2$  satisfy  $q_{20}: \top$ , the algorithm claims that the minimal rank of  $\{s_1, s_2\}$  is 20. ■

## Correctness of the Algorithm

We will now show that algorithm MIN-RANK correctly determines the minimal rank of all states in  $G_P$ . We observe that the removal of edges departing from  $q_0$ -states does not modify the value of the minimal rank. That is, for every  $s \in G_P$ ,  $\mathcal{R}(s)$  is the minimal rank also when we only consider segments contained in  $G_P^-$  which originate at  $s$ . This is because if a segment  $\sigma$  has a prefix  $\sigma_1$  which terminates in a  $q_0$ -state,  $\sigma$  satisfies  $\psi_j$  iff  $\sigma_1$  does.

We also observe that all states that belong to the same MSCS  $N_i$  have the same rank. If  $\mathcal{R}(s) = 0$  for some  $s \in N_i$ , then  $s \Vdash q_0$  and  $s$  has no departing edges in  $G_P^-$ . Consequently,  $N_i$  consists of the single node  $s$ . Otherwise  $\mathcal{R}(s) > 0$  for all  $s \in N_i$ . Consider two distinct states  $s_1, s_2 \in N_i$ . Since  $N_i$  is strongly connected there is a path from  $s_1$  to  $s_2$  and another path from  $s_2$  to  $s_1$ . By transitive closure of property R3 of minimal ranks,  $\mathcal{R}(s_1) \geq \mathcal{R}(s_2)$  and  $\mathcal{R}(s_2) \geq \mathcal{R}(s_1)$ . It follows that  $\mathcal{R}(s_1) = \mathcal{R}(s_2)$ .

The proof proceeds to show by induction on  $i = d, d-1, \dots, 1$  that the number  $r_i$  computed in step 4 of the algorithm equals the minimal rank  $\mathcal{R}(s)$  for every  $s \in N_i$ .

### Base

For  $i = d$ ,  $N_d = N_d$  has no disjoint successors. Let  $s$  be some state in  $N_d$ . By the definition of  $r_i$  in step 4, all states in  $N_d$  satisfy  $q_{r_i}$ . Since the only segments originating in  $s$  stay contained in  $N_d$  they all satisfy  $\psi_{r_i}$ . It follows that  $r_i$  is a rank of  $s$ . Since  $\mathcal{R}(s)$  is the minimal rank of  $s$ ,  $r_i \geq \mathcal{R}(s)$ .

By property R2 all states in  $N_i$  satisfy  $q_{\mathcal{R}(s)}$ . Since  $r_i$  is the smallest index

$j \geq 0$  such that all states in  $N_i$  satisfy  $q_j$ , it follows that  $\mathcal{R}(s) \geq r_i$ .

Thus  $r_i = \mathcal{R}(s)$ .

### Induction Step

Assume that the induction hypothesis holds for  $N_{i+1}, \dots, N_d$ , and we currently compute  $r_i$  for mscs  $N_i$  with distinct successors  $N_{j_1}, \dots, N_{j_b}$ . If  $b = 0$ , i.e.,  $N_i$  has no disjoint successors, then the proof proceeds as in the base case. Otherwise, assume that  $N_i$  has at least one disjoint successor.

Let  $s$  be some state in  $N_i$ . We will show that  $r_i$  is a rank of  $s$ . Let  $\sigma$  be a segment originating at  $s$ . There are two cases:

If  $\sigma$  stays contained within  $N_i$  then all its states satisfy  $q_{r_i}$ . Consequently,  $\sigma \models \psi_{r_i}$ .

Otherwise,  $\sigma$  can be decomposed into  $\sigma = \sigma_i \cdot \sigma_j$  where  $\sigma_i$  is contained within  $N_i$  and the first state in  $\sigma_j$  belongs to  $N_j$ , for some  $j \in \{j_1, \dots, j_b\}$ ,  $j > i$ . Since  $\sigma_j$  originates at an  $N_j$ -state, the induction hypothesis implies  $\sigma_j \models \psi_{r_j}$ . By the computation of  $r_i$ ,  $r_i \geq r_j$  and all states within  $\sigma$  satisfy  $q_{r_i}$ . It follows that  $\sigma \models \psi_{r_i}$ .

Thus,  $r_i$  is a rank of  $s \in N_i$ . Since  $\mathcal{R}(s)$  is the minimal rank of  $s$ , we have  $r_i \geq \mathcal{R}(s)$ .

Next, we observe that, by property R2, all states in  $N_i$  satisfy  $q_{\mathcal{R}(s)}$ . Furthermore, by property R3,  $\mathcal{R}(s) \geq \mathcal{R}(N_j)$  for every  $N_j$ , a disjoint successor of  $N_i$ , where  $\mathcal{R}(N_j)$  denotes the common minimal rank of all states belonging to  $N_j$ . By the induction hypothesis,  $\mathcal{R}(N_j) = r_j$ . Since  $r_i$  is the minimal index  $k$  such that all  $N_i$ -states satisfy  $q_k$  and  $k \geq r_j$  for every  $N_j$ , a disjoint successor of  $N_i$ , it follows that  $\mathcal{R}(s) \geq r_i$ .

We conclude that  $r_i = \mathcal{R}(s)$ .

### Checking for $P$ -Validity

Assume that we wish to check whether the formula

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0$$

is valid over a given finite-state program  $P$ .

This can be done by the following algorithm:

**Algorithm CHECK-NWAIT** — checking whether a nested waiting-for formula is  $P$ -valid

To check whether formula

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0$$

is valid over finite-state program  $P$ , perform the following steps:

- *Construction and Rank Determination* — Use Algorithm MIN-RANK to construct the state-transition graph  $G_P$  corresponding to  $P$ , and determine the rank  $\mathcal{R}(s)$  for all states  $s$  in  $G_P$ .
- *Validation* — For each  $p$ -state  $s$  in  $G_P$  check that  $\mathcal{R}(s) \leq m$ .

The nested waiting-for formula is  $P$ -valid iff all  $p$ -states have a rank smaller than  $m + 1$ .

**Example** Consider a transition system SUB2 with the following components:

*System variables:*  $\{y: \text{integer}\}$ .

*Initial condition:*  $\Theta: -3 \leq y \leq -1$ .

*Transitions:*  $\tau = \{\tau_I, \tau_<, \tau_2, \tau_3\}$  with corresponding transition relations

$$\begin{array}{ll} \rho_I: & y' = y \\ \rho_<: & y < 0 \wedge -3 \leq y' \leq 8 \\ & \end{array} \quad \begin{array}{ll} \rho_2: & y \geq 0 \wedge y' = y - 2 \\ \rho_3: & y \geq 0 \wedge y' = y - 3. \end{array}$$

The state-transition graph  $G_{\text{SUB2}}$  for this finite-state program is presented in Fig. 3.20, where we have omitted the self-edges induced by the idling transition  $\tau_I$ . Following our encapsulation conventions, the edge labeled with  $\tau_<$  represents  $\tau_<$ -labeled edges connecting each of  $s_{-3}, s_{-2}$ , and  $s_{-1}$  to all 12 states in the system.

We are interested in checking whether the formula

$$\psi: \underbrace{y \leq 8}_{p} \Rightarrow \underbrace{\text{even}(y)}_{q_3} \mathcal{W} \underbrace{\text{odd}(y)}_{q_2} \mathcal{W} \underbrace{\text{even}(y)}_{q_1} \mathcal{W} \underbrace{y < 0}_{q_0}$$

$\psi_3$

is valid over system SUB2.

The graph  $G_P^-$  is obtained by removing all the  $\tau_<$  labeled edges. Decomposing  $G_P^-$  into MSCS, we discover that each state is an MSCS and their sorting order can be given by

$$s_8, \quad s_7, \quad s_6, \quad s_5, \quad s_4, \quad s_3, \quad s_2, \quad s_1, \quad s_0, \quad s_{-1}, \quad s_{-2}, \quad s_{-3}.$$

Computing minimal ranks by algorithm MIN-RANK, we observe that the algorithm yields

$$r_{-3} = r_{-2} = r_{-1} = 0,$$

since each of  $s_{-3}, s_{-2}, s_{-1}$  is a  $q_0$ -state whose MSCS has no disjoint successors in  $G_P^-$ . We proceed to compute as follows:

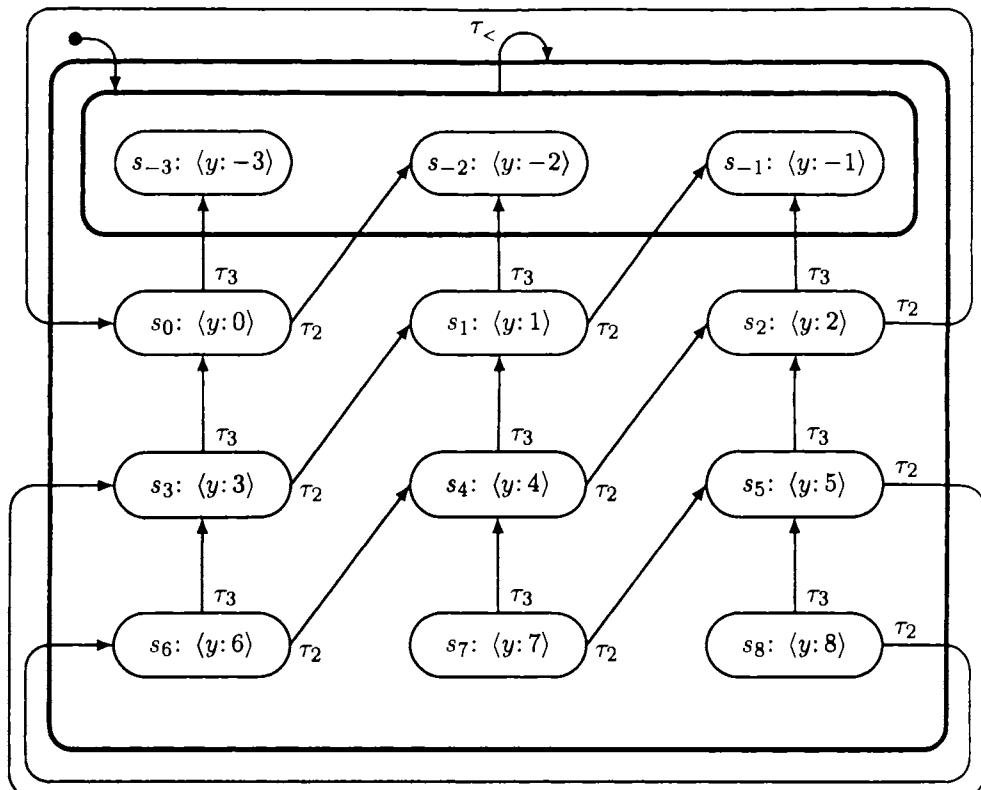


Fig. 3.20. State-transition graph for SUB2.

$$\begin{aligned}
 r_0 &= \text{lowest}(\{s_0\}, \max\{0, 0\}) = \text{lowest}(\{s_0\}, 0) = 1 \\
 r_1 &= \text{lowest}(\{s_1\}, 0) = 2 \quad \text{since } s_1 \nVdash q_1: \text{even}(y) \\
 r_2 &= \text{lowest}(\{s_2\}, \max\{0, 1\}) = \text{lowest}(\{s_2\}, 1) = 1 \\
 r_3 &= \text{lowest}(\{s_3\}, 2) = 2 \\
 r_4 &= \text{lowest}(\{s_4\}, 2) = 3 \\
 r_5 &= \text{lowest}(\{s_5\}, 2) = 2 \\
 r_6 &= \text{lowest}(\{s_6\}, 3) = 3 \\
 r_7 &= \text{lowest}(\{s_7\}, 3) = 4 \quad \text{since } s_7 \nVdash q_3: \text{even}(y) \\
 r_8 &= \text{lowest}(\{s_8\}, 3) = 3.
 \end{aligned}$$

It follows that segments departing from all states, except for  $s_7$ , satisfy  $\psi_3$ .  
The segment

$$s_7: \langle y: 7 \rangle \xrightarrow{\tau_3} s_4: \langle y: 4 \rangle \xrightarrow{\tau_3} s_1: \langle y: 1 \rangle$$

does not satisfy  $\psi_3$ . It follows that system SUB2 does not satisfy property  $\psi$  since there exists a computation  $\hat{\sigma}$ , starting with the prefix

$$s_{-1}: \langle y: -1 \rangle, \quad s_7: \langle y: 7 \rangle, \quad s_4: \langle y: 4 \rangle, \quad s_1: \langle y: 1 \rangle,$$

such that position 1 satisfies  $p: y \leq 8$  but does not satisfy  $\psi_3$ . ■

### Example (Peterson's algorithm — version 2)

As our final example, we consider program MUX-PET2 presented in Fig. 3.7 (page 268). We would like to check the validity over MUX-PET2 of the 2-bounded overtaking formula

$$\begin{array}{lcl} at\_l_i \Rightarrow & \underbrace{\neg at\_m_{5,6}}_{q_5} \mathcal{W} & \underbrace{at\_m_{5,6}}_{q_4} \mathcal{W} & \underbrace{\neg at\_m_{5,6}}_{q_3} \mathcal{W} & \underbrace{at\_m_{5,6}}_{q_2} \mathcal{W} \\ & q_5 & q_4 & q_3 & q_2 \\ & & & & \\ & & \underbrace{\neg at\_m_{5,6}}_{q_1} \mathcal{W} & \underbrace{at\_l_{5,6}}_{q_0} & \end{array}$$

for various values of  $i$ .

In Fig. 3.21 we present the graph  $G_P^-$  obtained by constructing the state-transition graph of program MUX-PET2 (see Fig. 3.7) and deleting all edges departing from the  $q_0$ -states  $s_6$ ,  $s_{10}$ , and  $s_{19}$ . To reduce the number of nodes appearing in the graph, we merged several states appearing in the full graph into nodes, such as  $s_0$ , which represent several states. For example, node  $s_0$  represents 9 states.

A sorted decomposition of this graph is given by

$$\begin{aligned} & \{s_0\}, \{s_1\}, \{s_2\}, \{s_4\}, \{s_5, s_9, s_{13}, s_{16}\}, \{s_{12}\}, \{s_{15}\}, \{s_{17}\}, \\ & \{s_8\}, \{s_{18}\}, \{s_{14}\}, \{s_3\}, \{s_7\}, \{s_{11}\}, \{s_6\}, \{s_{10}\}, \{s_{19}\}. \end{aligned}$$

Proceeding backwards in this list, we assign ranks as follows

$$\begin{array}{llll} r_{19} = r_{10} = r_6 & = 0 \\ r_{11} = r_7 = r_3 & = 1 \\ r_{14} & = 2 \\ r_{18} = r_8 = r_{17} = r_{15} & = 3 \\ r_{12} & = 4 \\ r_5 = r_9 = r_{13} = r_{16} & = 6 \\ r_4 & = 3 \\ r_2 & = 6 \\ r_1 & = 3 \\ r_0 & = 6. \end{array}$$

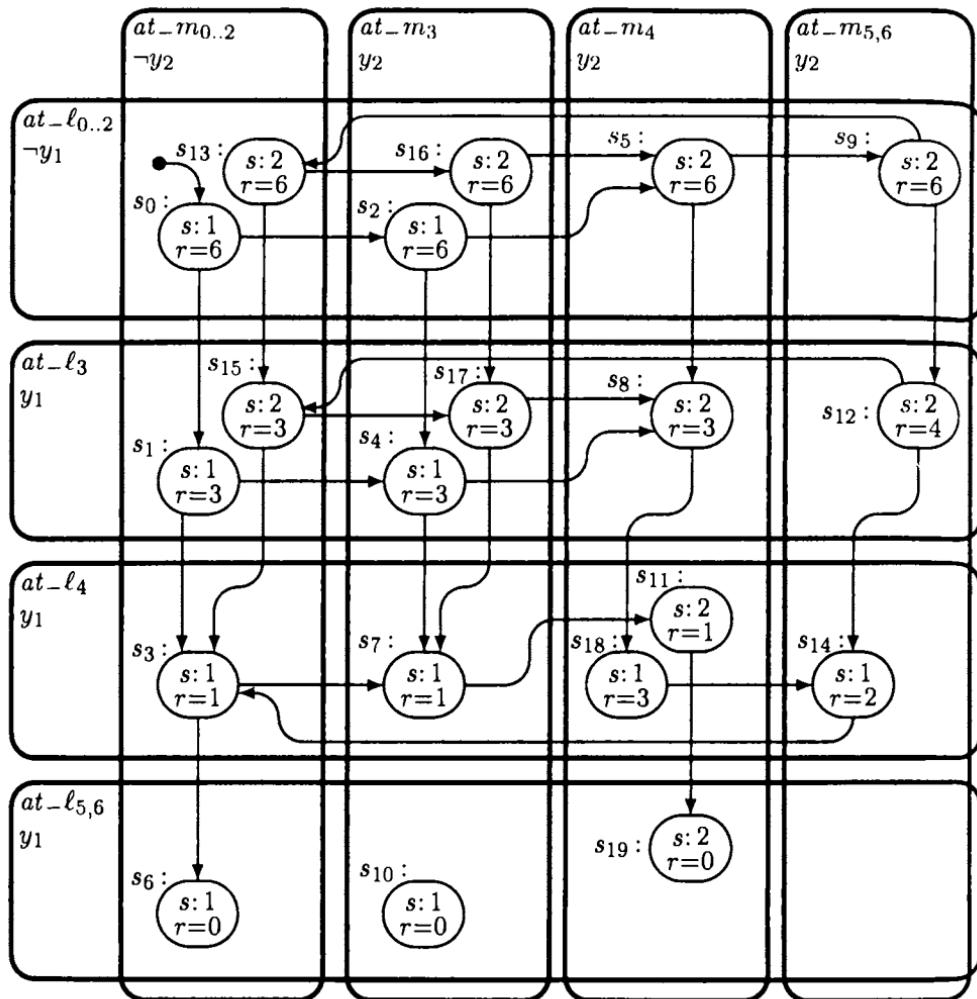


Fig. 3.21. Graph  $G_P^-$  for program MUX-PET2.

Let

$$\psi_5: q_5 \mathcal{W} q_4 \cdots q_1 \mathcal{W} q_0.$$

We observe that all states satisfying  $at\_l_{0..2}$  have been assigned the minimal rank 6. For example, states  $s_0$ ,  $s_2$ ,  $s_5$ ,  $s_9$ ,  $s_{13}$ , and  $s_{16}$  are such states. Consequently,  $at\_l_{0..2} \Rightarrow \psi_5$  is not  $P$ -valid.

On the other hand, all states satisfying  $at\_l_{3..6}$  have ranks smaller than 6. We conclude that

$$\underbrace{at\_l_{3..6} \Rightarrow}_{\psi_5} \underbrace{(\neg at\_m_{5,6}) W at\_m_{5,6} W (\neg at\_m_{5,6}) W at\_m_{5,6} W (\neg at\_m_{5,6}) W at\_l_{5,6}}_{\psi_5}$$

is  $P$ -valid. ■

## Problems

### Problem 3.1 (reflexive versions of the rules) page 257

The basic rule for invariance, rule INV-B (Fig. 1.1, page 87), has the property that if the conclusion of the rule is  $P$ -valid, then each of the premises is  $P$ -state valid. This property is not shared by rule WAIT-B (Fig. 3.1).

- (a) Give an example of a program  $P$  and two assertions,  $\varphi$  and  $\psi$ , such that the formula  $\varphi \Rightarrow \varphi W \psi$  is  $P$ -valid, yet the verification condition  $\{\varphi\} T \{\varphi \vee \psi\}$  is not  $P$ -state valid.

An alternative basic rule is rule REFL-B-WAIT, which can be written as follows:

$$\frac{\{\varphi \wedge \neg\psi\} T \{\varphi\}}{\varphi \Rightarrow \varphi W \psi}$$

- (b) Show that rule REFL-B-WAIT is sound.  
 (c) Show that if  $\varphi \Rightarrow \varphi W \psi$  is  $P$ -valid then the premise of rule REFL-B-WAIT is  $P$ -state valid.  
 (d) Show that every waiting-for formula that can be proven by rule B-WAIT can also be proven by rule REFL-B-WAIT but not vice versa.

In a similar way, there exists a reflexive version of rule WAIT, rule REFL-WAIT, which is presented in Fig. 3.22.

For assertions  $p, q, r, \psi$ ,

$$\begin{array}{l} W1. \quad p \rightarrow \psi \vee r \\ \widehat{W2.} \quad \psi \rightarrow q \vee r \\ \widehat{W3.} \quad \{\psi \wedge \neg r\} T \{\psi\} \\ \hline p \Rightarrow q W r \end{array}$$

Fig. 3.22. Rule REFL-WAIT (reflexive version of WAIT).

- (e) Show that rule REFL-WAIT is sound.
- (f) Consider a program  $P$  and assertions  $p$ ,  $q$ , and  $r$ . Show that there exists an assertion  $\varphi$  such that premises W1–W3 of rule WAIT are state valid iff there exists an assertion  $\psi$  such that premises  $W_1$ ,  $\widehat{W}_2$ , and  $\widehat{W}_3$  of rule REFL-WAIT are state valid. Show the same when the premises are required to be  $P$ -state valid.

This shows that, while rule REFL-B-WAIT is stronger than rule B-WAIT, i.e., it can prove more *waiting-for* formulas, rule REFL-WAIT has a proving power equal to that of rule WAIT.

**Problem 3.2** (a rule for the precedence operator) page 264

In Problem 0.8, we defined the precedence operator in terms of the waiting-for operator. Based on that definition, we can construct a proof rule, called PREC, to prove formulas of the form  $p \Rightarrow (q \mathcal{P} r)$ . Construct the proof rule PREC in the style of rule WAIT.

**Problem 3.3** (using rule PREC) page 264

Use rule PREC, formulated in Problem 3.2, to prove the property

$$at_{-\ell_4} \wedge at_{-m_0..3} \Rightarrow at_{-\ell_5} \mathcal{P} at_{-m_5}.$$

for Program MUX-PET2 of Fig. 3.7 (page 268). This formula states that, from the time Process  $P_1$  is at location  $\ell_4$  while Process  $P_2$  is still at the location range  $m_0, \dots, m_3$ , Process  $P_1$  will precede Process  $P_2$  in getting to the critical section.

**Problem 3.4** (unconditional nested waiting-for) page 267

- (a) Show that the formula

$$\psi_1: p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0$$

is equivalent to the formula

$$\square((\neg p) \mathcal{W} q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0),$$

which can also be written as

$$\psi_2: T \Rightarrow (\neg p) \mathcal{W} q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0.$$

- (b) Show that a proof of the  $P$ -validity of  $\psi_1$  can be translated into a proof of the  $P$ -validity of  $\psi_2$  and vice versa. That is, given intermediate assertions  $\varphi_0, \dots, \varphi_m$  that satisfy the premises of rule NWAIT for a proof of  $\psi_1$ , we can use them to construct assertions  $\chi_0, \dots, \chi_{m+1}$  that satisfy the premises of rule NWAIT for a proof of  $\psi_2$ . Symmetrically, given  $\chi_0, \dots, \chi_{m+1}$  that satisfy the premises for  $\psi_2$ , we can use them to construct assertions  $\varphi_0, \dots, \varphi_m$  that satisfy the premises for  $\psi_1$ .

**Problem 3.5** (Dekker's algorithm) page 272

Analyze the overtaking properties for program MUX-DEK of Fig. 1.15 (page 126).

- Show that, from a state satisfying  $at_{-l_{3,4}} \wedge t = 1$ ,  $P_2$  can visit  $m_8$  at most once ahead of  $P_1$  visiting  $\ell_8$ .
- What can be said about overtaking from states satisfying  $at_{-\ell_{0..2}}$ ?
- What can be said about overtaking from states satisfying  $at_{-\ell_{5..7}} \vee (at_{-l_{3,4}} \wedge t = 2)$ ?

**Problem 3.6** (Dekker's algorithm, variant B) page 272

To improve the overtaking properties of Dekker's Algorithm, we suggest program MUX-DEK-B of Fig. 3.23, which is a modified version of program MUX-DEK. The main differences between the two programs are that variables  $y_1$  and  $y_2$  in program MUX-DEK-B range over  $\{0..2\}$ , and that two new statements at locations  $\ell_{10}$  and  $m_{10}$  cause each process to wait until variable  $y_i$  of its rival assumes a value different from 1.

**local**  $y_1, y_2, t$ : integer **where**  $y_1 = 0, y_2 = 0, t = 1$

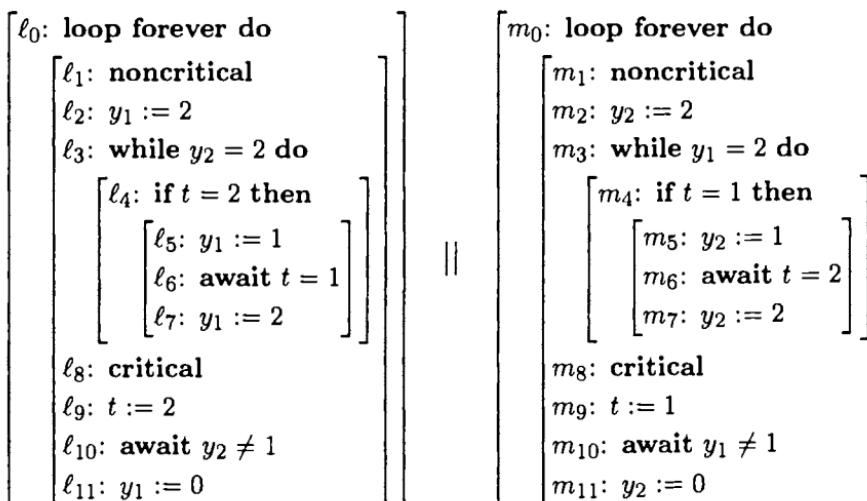


Fig. 3.23. Program MUX-DEK-B.

Analyze the overtaking properties of program MUX-DEK-B. Prove that overtaking of  $P_1$  by  $P_2$  from states satisfying  $at_{-l_{3..7}}$  is bounded. What is the bound?

**Problem 3.7** (bakery algorithm, version A) page 272

Analyze the overtaking properties of program MUX-BAK-A of Fig. 1.33 (page 159). Show that, from the time  $P_1$  is at  $\ell_3$ ,  $P_2$  may visit  $m_4$  ahead of  $P_1$  visiting  $\ell_4$  at most once.

**\* Problem 3.8** (bakery algorithm, version c) page 272

Analyze the overtaking properties of program MUX-BAK-C of Fig. 1.35 (page 160). Characterize the states for which a bound on the amount of overtaking can be given, and specify the bounds for each class of states.

**Problem 3.9** (INVARIANCE diagrams) page 280

Construct an invariance verification diagram that proves mutual exclusion for the following programs:

- (a) Program MUX-DEK of Fig. 1.15 (page 126).
- (b) Program MUX-DEK-A of Fig. 1.29 (page 156).
- (c) Program MUX-VAL-3 of Fig. 1.31 (page 157).
- (d) Program MUX-BAK-A of Fig. 1.33 (page 159).
- (e) Program MUX-BAK-C of Fig. 1.35 (page 160).
- (f) Program MUX-SWAP of Fig. 1.36 (page 161).
- (g) Program MUX-TESTSET of Fig. 1.37 (page 162).

**Problem 3.10** (WAIT diagrams) page 280

Construct a WAIT diagram for the overtaking property in the following problems:

- (a) Problem 3.5.
- (b) Problem 3.6.
- (c) Problem 3.7.
- (d) Problem 3.8.

**Problem 3.11** (program MUX-SWAP) page 280

Analyze the overtaking properties for program MUX-SWAP of Fig. 1.36 (page 161). In particular:

- (a) What can be said about overtaking from states which satisfy  $at\_l_2$ ?
- (b) What can be said about overtaking from states which satisfy  $at\_l_{3..4}$ ?
- (c) Construct the WAIT diagram for your answer to part (b).

**Problem 3.12** (shared-variables resource allocator) page 287

Program RES-SV, presented in Fig. 3.24, manages the allocation of a single resource between  $M$  customer processes which use shared variables to communicate with the allocator process. This is a shared-variables version of program RES-MP, presented in Fig. 3.15 (page 282).

in  $M$  : integer where  $M > 0$   
 local  $g, r$ : array [1.. $M$ ] of boolean where  $g = F, r = F$

$A :: \left[ \begin{array}{l} \text{local } t: \text{integer where } t = 1 \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{if } r[t] \text{ then} \\ \quad \left[ \begin{array}{l} m_2: g[t] := T \\ m_3: \text{await } \neg r[t] \\ m_4: g[t] := F \end{array} \right] \\ m_5: t := t \oplus_M 1 \end{array} \right] \end{array} \right]$

$\parallel$   
 $\left[ \begin{array}{l} M \\ \prod_{i=1}^M C[i] :: \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: r[i] := T \\ \ell_3: \text{await } g[i] \\ \ell_4: \text{critical} \\ \ell_5: r[i] := F \\ \ell_6: \text{await } \neg g[i] \end{array} \right] \end{array} \right]$

Fig. 3.24. Program RES-SV (resource allocator) --  
 shared-variables version.

This program was studied in Section 2.2, where we established for it mutual exclusion and some additional invariants. The property of 1-bounded overtaking for program ALLOCATOR-SV is stated by the formula

$$\psi: at\_l_3[k] \Rightarrow (\neg at\_l_{4,5}[n]) \mathcal{W} at\_l_{4,5}[n] \mathcal{W} (\neg at\_l_{4,5}[n]) \mathcal{W} at\_l_{4,5}[k]$$

for every  $k, n \in [1..M]$ ,  $k \neq n$ .

Prove that  $\psi$  is valid over program RES-SV. In your proof, you may use any of the invariants stated in Section 2.2.

**Problem 3.13** (shared-variables allocator with a single array) page 287

Consider program RES-SV-S, presented in Fig. 3.25. This shared-variables program improves upon program RES-SV in using only a single shared array for communication.

```

in      M:integer where M > 0
local u :array [1..M] of boolean
      where u[j] = F for 1 ≤ j ≤ M

A ::   
$$\left[ \begin{array}{l} \text{local } t: \text{integer where } t = 1 \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{if } u[t] \text{ then} \\ \quad \left[ \begin{array}{l} m_2: u[t] := \text{F} \\ m_3: \text{await } u[t] \\ m_4: u[t] := \text{F} \end{array} \right] \\ m_5: t := t \oplus_M 1 \end{array} \right] \end{array} \right]$$

||

i=1   
$$\left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: u[i] := \text{T} \\ \ell_3: \text{await } \neg u[i] \\ \ell_4: \text{critical} \\ \ell_5: u[i] := \text{T} \\ \ell_6: \text{await } \neg u[i] \end{array} \right] \\ M \end{array} \right]$$

      || C[i] ::
```

Fig. 3.25. Program RES-SV-S (resource allocator with a single communication array).

- (a) Show that this program maintains mutual exclusion, which can be stated by the assertion

$$\neg(at\_l_4[k] \wedge at\_l_4[n]),$$

claimed for every  $k, n \in [1..M]$ ,  $k \neq n$ .

(b) Prove the property of 1-bounded overtaking which can be stated by the nested *waiting-for* formula

$$at\_l_3[k] \Rightarrow (\neg at\_l_{4,5}[n] \mathcal{W} at\_l_{4,5}[n] \mathcal{W} (\neg at\_l_{4,5}[n]) \mathcal{W} at\_l_{4,5}[k]),$$

claimed for every  $k, n \in [1..M], k \neq n$ .

**Problem 3.14** (program PROD-CONS) page 287

Program PROD-CONS of Fig. 3.26 represents a producer-consumer program that communicates by asynchronous channels.

$\text{local } send, ack: \text{channel } [1..] \text{ of integer}$ $\text{where } send = \Lambda, ack = \underbrace{[1, \dots, 1]}_N$		
$Prod :: \left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: ack \Rightarrow t \\ \ell_3: send \Leftarrow x \end{array} \right] \end{array} \right] \parallel Cons :: \left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: send \Rightarrow y \\ m_2: ack \Leftarrow 1 \\ m_3: \text{consume } y \end{array} \right] \end{array} \right]$		

Fig. 3.26. Program PROD-CONS: producer-consumer with asynchronous channels.

In Section 2.1, we considered this program and proved that asynchronous channel *send* never carries more than  $N$  messages at the same time.

Here we are interested in another property of this program, the property stating that every value consumed at  $m_3$  must have been previously produced at  $\ell_1$ . We define the following abbreviations:

$$prod(u): at\_l_2 \wedge x = u \quad cons(u): at\_m_3 \wedge y = u.$$

The property connecting consumption to production can be expressed by the *waiting-for* formula

$$\varphi: \Theta \Rightarrow (\neg cons(u)) \mathcal{W} prod(u).$$

This formula states that, starting at the beginning of any computation ( $\Theta$  characterizes the initial state), we cannot observe a state in which  $u$  is consumed before we observe a state in which  $u$  is produced.

Prove that formula  $\varphi$  is valid over program PROD-CONS. In your proof you may use the expression  $u \in send$  to indicate a state in which  $u$  is one of the values queued in channel *send*.

**Problem 3.15** (completeness by a backward view) page 296

The proof of completeness presented in Section 3.6 based its construction of the assertions  $\varphi_1, \dots, \varphi_m$  on a *forward view*. That is, a state  $s$  was assigned a rank  $j$  based on the computation segments that may originate at  $s$ .

A dual approach is based on a *backward view* observing the computation segments that may lead to  $s$ .

A segment  $s_a, \dots, s_b$  is called *q<sub>0</sub>-free* if no  $s_i$ ,  $i = a, \dots, b$ , satisfies  $q_0$ . A segment  $s_a, \dots, s_b$  is said to have *height*  $j \leq m$  if it satisfies the formula  $q_m \mathcal{W} q_{m-1} \dots q_j \mathcal{W} F$ . It is said to have *maximal height*  $j$  if it has height  $j$  and there is no  $j'$ ,  $j < j' \leq m$ , such that  $s_a, \dots, s_b$  has height  $j'$ .

A state  $s$  is defined to be a *j-state*, for  $1 \leq j \leq m$ , if there exists a  $P$ -accessible state  $\hat{s}$  satisfying  $p$  and a  $q_0$ -free segment

$$\hat{s} = s_a, \dots, s_b = s$$

of maximal height  $j$ .

For each  $j$ ,  $1 \leq j \leq m$ ,  $\varphi_j$  is intended to characterize the set of  $j$ -states. As before, we take  $\varphi_0$  to be  $q_0$ .

- (a) Show that if  $p \Rightarrow q_m \mathcal{W} q_{m-1} \dots q_1 \mathcal{W} q_0$  is  $P$ -valid then the premises of rule NWAIT are  $P$ -state valid.
- (b) Write formulas that may use dynamic arrays to define the assertions  $\varphi_i$ , for  $i = 1, \dots, m$ .

## Bibliographic Remarks

**Precedence properties:** Temporal specification of precedence properties was proposed in Manna and Pnueli [1981a], using the *precedence* operator  $\mathcal{P}$  considered in Problem 0.8. This operator was used to specify the property of “absence of unsolicited response.” The waiting-for operator, under the name *unless*, was first introduced in Manna and Pnueli [1983b]. In fact, much of Chapter 3, including the proof rules for waiting-for and nested waiting-for formulas, the use of nested waiting-for formulas to express bounded overtaking, and the basic ideas of the completeness result established in Section 3.5, is based on Manna and Pnueli [1983b].

Lamport [1985b] points out an intrinsic difficulty in specifying precedence, using mutual exclusion as a running example. The problem can be viewed as a difficulty

in identifying the precise situations in which priority of one process over the other is established. In some respects, our notion of bounded overtaking and the discussion of programs with the bounded-overtaking property (page 280) is a possible solution to this difficulty.

**Verification diagrams:** Verification diagrams were first introduced to represent proofs of response properties such as in Manna and Pnueli [1983c]. Owicki and Lamport [1982] refer to them as proof lattices. Verification diagrams for precedence properties were first considered in Manna and Pnueli [1983b]. The encapsulation conventions adopted for verification diagrams are inspired by the Statecharts visual language introduced by Harel [1987]. A comprehensive presentation of the various types of verification diagrams corresponding to the different property classes and their associated rules is given in Manna and Pnueli [1994].

**Decomposition into MSCS's:** In the text, we use algorithm DECOMPOSE for decomposing a directed graph into a sequence of strongly connected components. There are two algorithms in the literature which provide this functionality with the optimal complexity of  $\mathcal{O}(V+E)$ . The first is the algorithm presented in Tarjan [1972]. Another algorithm with the same complexity is described in Cormen, Leiserson, and Rivest [1990], where it is called the “strongly-connected-components” algorithm.



## Chapter 4

# General Safety

We have defined a safety property to be any property that can be specified by a formula of the form  $\square p$  for some past formula  $p$ .

In Chapter 1, we considered the special case that  $p$  is a state formula (assertion), referring to the properties specifiable by such formulas as invariance properties.

In Chapter 3, we considered an extension of the class of invariance properties, called precedence properties. These are the properties that can be expressed by a nested waiting-for formula of the form

$$p \Rightarrow q_1 \mathcal{W} q_2 \cdots q_m \mathcal{W} r.$$

As shown in Section 4.8, even this extension is still only a subclass of the general class of safety properties.

In this chapter we present rules that support the verification of the most general safety property.

We start, in Section 4.1, by showing how rule INV, applicable so far only for proving invariances of the form  $\square p$  for state formulas  $p$ , can be generalized to prove general safety formulas  $\square p$ , where  $p$  is a past formula. We then illustrate in Section 4.2 the use of the generalized rule INV-P for proving properties such as absence of unsolicited response for the producer-consumer program.

Past formulas are very useful for formulating assumptions about the environment. This capability is essential for carrying out compositional verification. In Section 4.3 we present a proof rule for compositional verification and illustrate its use.

In Section 4.4 we consider safety formulas of the special form  $p \Rightarrow \Diamond q$ , to which we refer as *causality formulas*. These formulas represent a causal link between the present occurrence of  $p$  and a past occurrence of  $q$ . For example, the property of absence of unsolicited response is naturally expressed by such a formula. We present a specially tailored rule for establishing such properties.

We then explore in Section 4.5 a verification methodology that breaks the proof of a formula  $p \Rightarrow \Diamond r$  into two proofs of the form  $p \Rightarrow \Diamond q$  and  $q \Rightarrow \Diamond r$ . We refer to this methodology as *backward analysis* and illustrate its use for proving a safety property  $\Box \neg p$  by deriving a contradiction, i.e., showing  $p \Rightarrow \Diamond F$ .

In Section 4.6 we consider proofs of order-preservation properties, showing, for example, that messages in the producer-consumer program are consumed in the same order they are produced.

Section 4.7 shows that many proofs of general safety properties, in particular proofs of order-preservation properties, can be simplified by using history variables. History variables are useful in many contexts and can often be used to replace past reasoning by state reasoning.

In Section 4.8, we consider properties expressible by nested back-to formulas. We point out the symmetry between nested back-to formulas and nested waiting-for formulas, and present a rule for nested back-to formulas.

Section 4.9 shows that rule INV-P is complete for proving safety formulas. Completeness of the rule means that whenever the conclusion is  $P$ -valid, it is possible to find an auxiliary formula  $\varphi$ , such that the rule's premises are  $P$ -valid.

Finally, Section 4.10 studies algorithmic verification of general safety formulas over finite-state programs.

## 4.1 Invariance Rule for Past Formulas

The rules we propose for proving invariance of past formulas are essentially the same rules we used for proving invariance of state formulas. As a matter of fact, we will show that most of the basic concepts on which we based our proof rules for state invariants can be extended to support a very similar theory for past invariants.

In Fig. 4.1 we present the invariance rule INV-P for past formulas, which can be viewed as the (past) extension of rule INV (Fig. 1.5, page 92).

There are several differences between rule INV and rule INV-P. The immediately apparent differences are the use of the entailment operator  $\Rightarrow$  in premise P1 replacing the simple implication  $\rightarrow$  appearing in premise I1 of rule INV, and the formula  $(\varphi)_0$  in premise P2, to which we refer as the *initial version* of formula  $\varphi$ .

Another difference between INV and INV-P is that, for the case of past formulas  $\varphi$  and  $\psi$ , the verification condition for  $\tau \in \mathcal{T}$  has the extended definition

$$\{\varphi\} \tau \{\psi\}: \quad \rho_\tau \wedge \varphi \Rightarrow \psi'.$$

We refer to  $\psi'$  as the *past primed version of  $\psi$* . Note that the extended verification condition also replaces implication by entailment. This definition naturally

For past formulas  $\varphi, p$ ,

$$\begin{array}{l} \text{P1. } \varphi \Rightarrow p \\ \text{P2. } \Theta \rightarrow (\varphi)_0 \\ \text{P3. } \{\varphi\} T \{\varphi\} \\ \hline \\ \square p \end{array}$$

Fig. 4.1. Rule INV-P (past general invariance).

extends to  $\{\varphi\} T \{\psi\}$  and  $\{\varphi\} T \{\psi\}$ , for a set of transitions  $T \subseteq \mathcal{T}$  and the entire set of transitions  $\mathcal{T}$ , respectively.

The following discussion considers these differences and explains why these changes are necessary for the more general case of past formulas.

### ***The Use of Entailments***

Compare premise I1 of rule INV, reading  $\varphi \rightarrow p$ , with premise P1 of rule INV-P, reading  $\varphi \Rightarrow p$ . In both cases the intended meaning of the premise is that  $\varphi$  should imply  $p$  at all positions of all computations. Writing P1 fully, it would read

$$P \models \square(\varphi \rightarrow p).$$

For the case that  $\varphi$  and  $p$  are state formulas, we know that  $\square(\varphi \rightarrow p)$  is  $P$ -valid iff  $\varphi \rightarrow p$  is  $P$ -state valid, i.e.,  $P \models \varphi \rightarrow p$ , which by our adopted rule convention can be written simply as

$$\varphi \rightarrow p.$$

For the case that  $\varphi$  or  $p$  is a past formula, the implication  $\varphi \rightarrow p$  only guarantees that  $\varphi$  implies  $p$  at position 0 but not necessarily at any other position. Consequently, we require  $P$ -validity of the entailment  $\varphi \Rightarrow p$ , i.e.,  $P \models \varphi \Rightarrow p$ , which by convention can be written simply as

$$\varphi \Rightarrow p.$$

### ***The Initial Version of a Past Formula***

Similarly to premise I2 of rule INV, premise P2 of rule INV-P provides the base case for the induction that underlies all invariance rules. It is intended to establish that the formula  $\varphi$  holds at the beginning of each computation. Some of the

information specific to the initial state is, of course, encapsulated in the initial condition  $\Theta$ , and this is why we have  $\Theta$  as the antecedent of the implication P2. However, for the evaluation of past formulas, the initial position possesses some additional unique characterizations that are not captured by  $\Theta$ . For example, the past formula  $\odot(x = 1)$ , where  $\odot$  is the *before* operator, is always true in the initial position, independent of  $\Theta$ .

To capture the initiality of the first position, P2 has the form

$$\Theta \rightarrow (\varphi)_0,$$

where  $(\varphi)_0$  is the *initial version* of the past formula  $\varphi$ . The initial version  $(\varphi)_0$  is a state formula that expresses the truth-value of  $\varphi$  at position 0. It is defined inductively as follows:

- For a variable  $x$ ,

$$(x)_0 = x.$$

- For an expression  $f(t_1, \dots, t_m)$ ,

$$(f(t_1, \dots, t_m))_0 = f((t_1)_0, \dots, (t_m)_0).$$

This also covers predicates and boolean connectives, such as  $\vee$  and  $\neg$ , by considering them as boolean expressions. Thus,

$$(p \vee q)_0 = (p)_0 \vee (q)_0, \quad (\neg p)_0 = \neg(p)_0.$$

- For a previous-value expression  $x^-$

$$(x^-)_0 = x.$$

- For a quantified formula

$$(\exists u: p)_0 = \exists u: (p)_0$$

$$(\forall u: p)_0 = \forall u: (p)_0.$$

- For the temporal operators

$$(\odot p)_0 = \text{F}, \quad (\odot p)_0 = \text{T}$$

$$(\Diamond p)_0 = (\Box p)_0 = (p)_0$$

$$(p \mathcal{S} q)_0 = (q)_0$$

$$(p \mathcal{B} q)_0 = (p)_0 \vee (q)_0.$$

For example, if we consider the past formula

$$p: \quad \Box(x > 5 \rightarrow \Diamond(x = 5)),$$

then its initial version is computed as follows:

$$\begin{aligned}
 p_0 &= (\Box(x > 5 \rightarrow \Diamond(x = 5)))_0 = (x > 5 \rightarrow \Diamond(x = 5))_0 \\
 &= (x > 5)_0 \rightarrow (\Diamond(x = 5))_0 = (x > 5)_0 \rightarrow (x = 5)_0 \\
 &= (x > 5) \rightarrow (x = 5) = x \leq 5.
 \end{aligned}$$

Note, in particular, that for the formula *first*, defined as  $\neg \Theta \top$ , we have

$$(\text{first})_0 = \top.$$

### The Primed Version of a Past Formula

Another important concept that we wish to generalize is the notion of the primed version of a formula. Recall that the intended meaning of  $\psi'$  is the value of  $\psi$  in the next state where, by convention, we refer to the values of variables in the next state by their primed version.

For example, if  $\psi$  is the assertion  $x = 5$ , then its primed version is  $x' = 5$ , telling us that  $\psi$  will hold in the next state if  $x'$ , the value of  $x$  in the next state, equals 5. Thus, for the simple case that  $\psi$  is a state formula, the primed version of  $\psi$  is obtained by replacing all free occurrences of the system variables in  $\psi$  by their primed version.

For the more general case, where  $\psi$  is a past formula, the situation is more complex. Consider, for example, the formula  $\psi: \Theta(x = 5)$ . It is obvious that  $\Theta(x = 5)$  holds in the next state iff  $x = 5$  holds in the current state. This shows that the primed version of  $\psi$  should be given by

$$\psi': (\Theta(x = 5))' = (x = 5).$$

In general,  $\psi'$  is a formula that expresses the truth-value of  $\psi$  at the next position. Formula  $\psi'$  must be expressed in terms of state and past formulas evaluated at the current position, and primed variables  $x'$  which denote the value of  $x$  at the next position.

Following is an inductive definition of  $\psi'$ :

- For a rigid variable  $u$  (cannot belong to  $V$ ),  
 $u' = u$ .
- The primed version of a flexible variable  $v$  is simply written as  $v'$ .
- For an expression  $f(t_1, \dots, t_m)$ ,

$$(f(t_1, \dots, t_m))' = f(t'_1, \dots, t'_m).$$

This also covers predicates and boolean connectives, such as  $\vee$  and  $\neg$ , by considering them as boolean expressions. For example, according to this definition,

$$(P(x) \vee Q(y, z))' = P(x') \vee Q(y', z'),$$

for  $x, y, z \in V$ .

- For a quantified formula,

$$(\exists u: \psi)' = \exists u: \psi'$$

$$(\forall u: \psi)' = \forall u: \psi'.$$

Up to this point, the definition says that “priming” distributes over all functions, predicates, and non-temporal connectives. These clauses are sufficient to cover the case of state formulas and are equivalent to the  $\psi[V \mapsto V']$  definition. It is only the forthcoming clauses, dealing with the past operators, that extend the definition to cover arbitrary past formulas.

- For a previous-value expression  $x^-$ ,

$$(x^-)' = x.$$

For the past operators  $\ominus$ ,  $\oslash$ ,  $\lozenge$ ,  $\boxdot$ ,  $\mathcal{S}$ , and  $\mathcal{B}$ , we have

- $(\ominus p)' = (\ominus p)' = p$

Since, by definition, the next state always has a predecessor (the current state), there is no difference between  $(\ominus p)'$  and  $(\ominus p)$ .

- $(\lozenge p)' = p' \vee \lozenge p$

Thus,  $\lozenge p$  holds in the next state iff either  $p$  holds there or  $\lozenge p$  already holds currently.

- $(\boxdot p)' = p' \wedge \boxdot p$

Thus,  $\boxdot p$  holds in the next state iff  $p$  holds there and  $\boxdot p$  holds currently.

- $(p \mathcal{S} q)' = q' \vee (p' \wedge p \mathcal{S} q)$

Thus,  $p \mathcal{S} q$  holds in the next state iff either  $q$  holds there, or  $p$  holds there and  $p \mathcal{S} q$  holds currently.

- $(p \mathcal{B} q)' = q' \vee (p' \wedge p \mathcal{B} q)$

Similarly,  $p \mathcal{B} q$  holds in the next state iff either  $q$  holds there, or  $p$  holds there and  $p \mathcal{B} q$  holds currently.

Note, in particular, that for the formula *first* we have (after simplification)

$$(\text{first})' = \text{F}.$$

**Example** Consider the formula

$$\psi: x > 5 \leftrightarrow \lozenge(x = 5).$$

Its primed version is given by

$$\begin{aligned} \psi' &= (x > 5 \leftrightarrow \lozenge(x = 5))' = (x > 5)' \leftrightarrow (\lozenge(x = 5))' \\ &= x' > 5 \leftrightarrow x' = 5 \vee \lozenge(x = 5). \end{aligned}$$

Note that this translation expresses the *next* value of  $\psi$ , i.e., the value of  $\psi$  in the next state, in terms of  $x'$ , the *next* value of the variable  $x$  and the *current* value of the temporal subformula  $\Diamond(x = 5)$ . ■

In order to establish the main property of primed past formulas, we use again the notion of a predictive computation.

Recall (page 84) that a computation is called *predictive* if each state  $s_j$  in the computation interprets each primed version  $x'$  of a system variable  $x \in V$  in the same way that  $s_{j+1}$  interprets  $x$ . That is,  $s_j[x'] = s_{j+1}[x]$ , for each  $x \in V$  and every  $j = 0, 1, \dots$ .

The following claim, which can be proved by induction on the structure of the past formula  $\psi$ , states that  $\psi'$  fulfills its intended role and predicts the value of  $\psi$  in the next position.

#### Claim 4.1 (past primed formulas)

For past formula  $\psi$ , predictive computation  $\sigma$ , and position  $j \geq 0$ ,  $(\sigma, j) \models \psi'$  if and only if  $(\sigma, j + 1) \models \psi$ .

**Example** Consider the following predictive computation of a program with a single system variable  $x$

$$\sigma: \quad s_0: \langle x: 0, x': 1 \rangle, s_1: \langle x: 1, x': 2 \rangle, s_2: \langle x: 2, x': 3 \rangle, s_3: \langle x: 3, x': 4 \rangle, \dots .$$

Let  $\psi$  be the past formula

$$\psi: \quad x = 3 \wedge \Diamond(x = 2) \wedge \Diamond\Diamond(x = 1).$$

The primed version of  $\psi$  is

$$\psi': \quad x' = 3 \wedge x = 2 \wedge \Diamond(x = 1).$$

It is straightforward to check that  $\psi'$  holds at position 2 and that (as stated by the claim)  $\psi$  holds at position 3. ■

### The Verification Condition

Next, we consider the generalized form of the *verification condition*.

Let  $\varphi$  and  $\psi$  be two past formulas, and  $\tau$  a transition of the program  $P$ . The verification condition of  $\varphi$  and  $\psi$ , relative to the transition  $\tau$ , is given by the formula

$$\{\varphi\} \tau \{\psi\}: \quad \rho_\tau \wedge \varphi \Rightarrow \psi',$$

where  $\psi'$  is the past primed version of  $\psi$ .

As with state formulas, the verification condition guarantees that  $\tau$  leads from a  $\varphi$ -position to a  $\psi$ -position. This is stated by the following proposition:

**Proposition 4.2** (past verification condition)

Let  $\{\varphi\} \tau \{\psi\}$  be a  $P$ -valid verification condition,  $\sigma$  be a  $P$ -computation, and  $i \geq 0$  be a position. If  $(\sigma, i) \models \varphi$  and  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ , then  $(\sigma, i + 1) \models \psi$ .

**Justification** Assume that  $\{\varphi\} \tau \{\psi\}$  is  $P$ -valid and let  $\sigma: s_0, s_1, \dots$  be a computation of  $P$  such that  $(\sigma, i) \models \varphi$  and  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ . To prove the proposition, we have to show that  $(\sigma, i + 1) \models \psi$ .

Let  $\tilde{\sigma}$  be the predictive version of  $\sigma$ . Since  $\sigma$  is a  $P$ -computation, so is  $\tilde{\sigma}$ . Since  $\{\varphi\} \tau \{\psi\}$  is  $P$ -valid, the implication  $\rho_\tau \wedge \varphi \rightarrow \psi'$  holds at all positions of all  $P$ -computations, in particular at position  $i$  of  $\tilde{\sigma}$ , i.e.,

$$(1) \quad (\tilde{\sigma}, i) \models \rho_\tau \wedge \varphi \rightarrow \psi'.$$

Since  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ ,  $\rho_\tau(s_i[V], s_{i+1}[V])$  is true. By the definition of predictive computations this implies that  $\rho_\tau(s_i[V], s_i[V'])$ , i.e.,

$$(2) \quad (\tilde{\sigma}, i) \models \rho_\tau.$$

As  $\varphi$  does not refer to  $V'$  and  $(\sigma, i) \models \varphi$ ,  $\varphi$  also holds at position  $i$  of  $\tilde{\sigma}$ , i.e.,

$$(3) \quad (\tilde{\sigma}, i) \models \varphi.$$

From (1), (2), and (3), we conclude that

$$(\tilde{\sigma}, i) \models \psi'$$

and, by Claim 4.1,

$$(\tilde{\sigma}, i + 1) \models \psi.$$

Since  $\psi$  does not refer to  $V'$ , it also holds at position  $i + 1$  of  $\sigma$ . It follows that

$$(\sigma, i + 1) \models \psi. \quad \blacksquare$$

**Example** Consider a case in which the past formulas are

$$\varphi: x = 2 \wedge \neg(x = 1)$$

$$\psi: x = 3 \wedge \neg(x = 2) \wedge \neg\neg(x = 1),$$

and the transition relation is

$$\rho_\tau: x' = x + 1.$$

The verification condition  $\{\varphi\} \tau \{\psi\}$  is given by the following entailment, which is obviously valid:

$$\underbrace{x' = x + 1}_{\rho_\tau} \wedge \underbrace{x = 2 \wedge \neg(x = 1)}_{\varphi} \Rightarrow \underbrace{x' = 3 \wedge x = 2 \wedge \neg(x = 1)}_{\psi'}.$$

Consider the computation

$$\sigma: \dots, s_1: \langle x: 1 \rangle, s_2: \langle x: 2 \rangle, s_3: \langle x: 3 \rangle, \dots$$

which satisfies  $\varphi$  at position 2, and in which  $s_3$  is a  $\tau$ -successor of  $s_2$ .

The predictive version of  $\sigma$  is

$$\tilde{\sigma}: \dots, \tilde{s}_1: \langle x: 1, x': 2 \rangle, \tilde{s}_2: \langle x: 2, x': 3 \rangle, \tilde{s}_3: \langle x: 3, x': 4 \rangle, \dots$$

We observe that  $\rho_\tau: x' = x + 1$  and  $\varphi: x = 2 \wedge \Theta(x = 1)$  hold at position 2 as does  $\psi'$ . As stated by the claim,  $\psi: x = 3 \wedge \Theta(x = 2) \wedge \Theta \Theta(x = 1)$  should hold at position 3 of  $\sigma$  (and  $\tilde{\sigma}$ ), which it does. ■

## The Idling Transition

Since the idling transition  $\tau_I$  is present in all transition systems derived from programs, we are repeatedly called to establish the validity of the verification condition

$$\rho_I \wedge \varphi \Rightarrow \varphi'.$$

In Chapter 1 (page 87), we observed that when  $\varphi$  is a state formula that only refers to system or rigid variables, the verification condition  $\rho_I \wedge \varphi \rightarrow \varphi'$  is trivially state valid because  $\tau_I$  modifies no system or rigid variables (and hence  $\varphi'$  is equivalent to  $\varphi$ ). We can express this fact by saying that such an assertion  $\varphi$  is always preserved by the idling transition.

This is no longer the case for general past formulas. Consider the following computation prefix:

$$\sigma: s_0: \langle x: 0 \rangle, s_1: \langle x: 1 \rangle, s_2: \langle x: 1 \rangle, \dots$$

where  $x$  is the only system variable. It is obvious that  $s_2$  is a  $\tau_I$ -successor of  $s_1$ . Consider the past formula  $\varphi: x = 1 \wedge \Theta(x = 0)$ . Obviously,  $(\sigma, 1) \models \varphi$  but  $(\sigma, 2) \not\models \varphi$ . This shows that past formulas are not, in general, preserved by the idling transition.

There is, however, a large class of past formulas that are preserved by the idling transition. These are past formulas which are *previous-free*, i.e., contain no subformulas of the form  $\Theta p$  or  $\odot p$ , or expressions of the form  $x^-$ . It is also required that these formulas contain no quantifiers and no flexible variables other than the system variables.

In **Problem 4.1**, the reader is asked to prove that previous-free past formulas are preserved under the idling transition.

## Establishing $P$ -Valid Entailments

The next question we address is how to establish the  $P$ -validity of the entailments appearing as premises of rule INV-P.

We present three methods for establishing the  $P$ -validity of entailments of the form  $p \Rightarrow q$  that are used as premises for rule INV-P.

- *Tautological Instantiation*

This method establishes validity of the entailment  $p[\alpha] \Rightarrow q[\alpha]$  by a temporal instantiation of a state-valid scheme  $p \rightarrow q$ . The resulting rule can be presented as

$$\frac{\models p \rightarrow q}{\models p[\alpha] \Rightarrow q[\alpha]}$$

where

$$\alpha: [p_1 \mapsto \varphi_1, \dots, p_r \mapsto \varphi_r]$$

is a temporal *instantiation (replacement)*. Such a replacement specifies a list of propositions  $p_1, \dots, p_r$  and a corresponding list of temporal formulas  $\varphi_1, \dots, \varphi_r$ . The instantiated formula  $p[\alpha]$  ( $q[\alpha]$ ) is obtained by replacing in  $p$  (in  $q$ ) each occurrence of  $p_i$  by the corresponding formula  $\varphi_i$ , for  $i = 1, \dots, r$ . It is required that the instantiation be *admissible*, i.e., no variable occurring in any of the  $\varphi_i$ 's is quantified in either  $p$  or  $q$ .

By this approach, we may derive the validity of

$$x \leq 5 \Rightarrow (x > 5 \rightarrow \Diamond(x = 5))$$

from the tautology (state-valid formula)

$$\neg p \rightarrow (p \rightarrow q),$$

using the replacement

$$\alpha: \left\{ \begin{array}{l} p \mapsto x > 5 \\ q \mapsto \Diamond(x = 5) \end{array} \right\}$$

We refer to this type of inference as *tautological instantiation*.

- *Rules INV and INV-P*

This method uses previous applications of rules INV and INV-P that establish  $P$ -validities of the form  $\Box p$ , i.e.,  $P \models \Box p$ . In the case that  $p$  has the form  $q \rightarrow r$ , the consequence is the entailment  $q \Rightarrow r$ .

As in the case of rule INV, this approach suggests a process of incremental verification, where previously derived invariances aid in the establishment of premises for the next application of rule INV-P.

- *Entailment Reasoning*

This method applies entailment reasoning to a valid entailment derived by tautological instantiation and a previously established  $P$ -valid formula of the form  $\Box \chi$ . It can be stated by the inference rule

$$\frac{\models \chi \wedge p \Rightarrow q \quad P \models \square \chi}{P \models p \Rightarrow q}$$

In the process of entailment reasoning, we may freely use the axioms and properties established for the temporal operators.

## 4.2 Applications of the Past Invariance Rule

In this section we illustrate the application of rule INV-P.

### Example (trivial system)

Consider the trivial transition system INC defined in Fig. 4.2.

$V:$	$\{x: \text{integer}\}$
$\Theta:$	$x = 0$
$T:$	$\{\tau_I, \tau\}$ where $\rho_\tau: x' = x + 1$
$\mathcal{J}:$	$\{\tau\}$
$C:$	$\{ \}$

Fig. 4.2. System INC.

Using rule INV-P, we wish to prove the following safety property for INC:

$$x = 10 \Rightarrow \Diamond(x = 5),$$

which can be viewed as the invariance of the formula

$$\psi: x = 10 \rightarrow \Diamond(x = 5).$$

The property states that any state in which  $x = 10$  must have been preceded by a state in which  $x = 5$ .

As is often the case, it is impossible to verify the inductiveness (i.e., satisfaction of P2 and P3) of  $\psi$ . We use a stronger auxiliary formula given by

$$\varphi: x > 5 \rightarrow \Diamond(x = 5).$$

Let us check that formulas  $\varphi$  and  $\psi$  satisfy premises P1–P3.

- Premise P1

Premise P1 reads

$$\underbrace{x > 5 \rightarrow \Diamond(x = 5)}_{\varphi} \Rightarrow \underbrace{x = 10 \rightarrow \Diamond(x = 5)}_{\psi}.$$

We establish it by tautological instantiation, applying the replacement

$$q \mapsto \Diamond(x = 5)$$

to the first-order valid implication

$$(x > 5 \rightarrow q) \rightarrow (x = 10 \rightarrow q).$$

- Premise P2

Premise P2 reads

$$\underbrace{x = 0}_{\Theta} \rightarrow \left( \underbrace{x > 5 \rightarrow \Diamond(x = 5)}_{\varphi} \right)_0.$$

Following the procedure for computing  $(x > 5 \rightarrow \Diamond(x = 5))_0$ , we obtain the implication  $x > 5 \rightarrow x = 5$ , which is equivalent to  $x \leq 5$ . Thus, premise P2 reduces to the obviously state-valid implication

$$x = 0 \rightarrow x \leq 5.$$

- Premise P3

Next, consider premise P3. Since formula  $\varphi$  is previous-free, it is sufficient to consider the verification condition for  $\tau$ , which is

$$\rho_\tau \wedge \underbrace{x > 5 \rightarrow \Diamond(x = 5)}_{\varphi} \Rightarrow \left( \underbrace{x > 5 \rightarrow \Diamond(x = 5)}_{\varphi} \right)'.$$

Let us compute  $(x > 5 \rightarrow \Diamond(x = 5))'$ . Following the inductive definition, we get

$$x' > 5 \rightarrow x' = 5 \vee \Diamond(x = 5),$$

equivalently

$$x' > 5 \rightarrow \Diamond(x = 5).$$

Thus, the verification condition amounts to

$$\underbrace{x' = x + 1}_{\rho_\tau} \wedge (x > 5 \rightarrow \Diamond(x = 5)) \Rightarrow (x' > 5 \rightarrow \Diamond(x = 5)).$$

Observing that  $\rho_\tau$  is  $x' = x + 1$ , it is sufficient to prove

$$(x > 5 \rightarrow \Diamond(x = 5)) \Rightarrow (x + 1 > 5 \rightarrow \Diamond(x = 5)).$$

Since  $x$  is an integer,  $x + 1 > 5$  implies that either  $x = 5$  or  $x > 5$ . Consequently, the above formula is equivalent (by propositional reasoning) to

$$(x > 5 \rightarrow \Diamond(x = 5)) \Rightarrow$$

$$(x = 5 \rightarrow \Diamond(x = 5)) \wedge (x > 5 \rightarrow \Diamond(x = 5)).$$

The right-hand side of this entailment consists of the conjunct  $x = 5 \rightarrow \Diamond(x = 5)$  which is always true, due to the reflexivity of the operator  $\Diamond$ , and the conjunct  $x > 5 \rightarrow \Diamond(x = 5)$ , which is identical to the antecedent of the entailment. Consequently, the entailment is  $P$ -valid.

The invariance of  $\psi$  follows by rule INV-P. ■

## Causality Properties

We define a *causality formula* to be a formula of the form

$$p \Rightarrow \Diamond r,$$

for past formulas  $p$  and  $r$ . Such a formula can express a causal link between  $r$  and  $p$ , stating that  $p$  cannot occur now unless it has been preceded by  $r$  in the past. We call such a formula a *state-causality formula* in the case that  $p$  and  $r$  are state formulas.

The causality formula  $p \Rightarrow \Diamond r$  states that a past occurrence of  $r$  is a necessary condition for  $p$ . One may argue that for  $r$  to be a real *cause* of  $p$ , it should also be a sufficient condition, ensuring that every  $r$  must be followed by a future  $p$ . In this chapter, we associate with causality only the “necessary condition” part. The “sufficient condition” part, requiring that every  $r$  is followed by a future  $p$ , is the main topic of the PROGRESS book.

We define the classes of *causality properties* and *state-causality properties* as the classes of properties expressible by formulas of the corresponding types. Since a causality formula can be written as  $\Box(p \rightarrow \Diamond r)$ , it is a canonical safety formula. It follows that the classes of causality and state-causality properties are all safety properties.

The property we have established above for the trivial transition system INC of Fig. 4.2 is given by the simple causality formula

$$\varphi: x = 10 \Rightarrow \Diamond(x = 5).$$

The problem we usually face in constructing a direct proof of a causality formula is that it connects two remote situations, without specifying anything about the computation segment separating them.

The only proof rule we currently have, rule INV-P, is based on tracing properties from one state to its successor and cannot perform jumps such as from  $x = 5$  to  $x = 10$ . This is why strengthening into inductive formulas is necessary. The strengthened formula characterizes the situations holding in the intermediate computation segment connecting the two remote situations.

In the previous example, we strengthened  $\psi$  into the inductive formula

$$\varphi: x > 5 \rightarrow \Diamond(x = 5),$$

which stated that not only at  $x = 10$  are we guaranteed to have  $x = 5$  in our past, but that this is guaranteed also for every state such that  $x > 5$ .

We propose and illustrate a basic principle, to which we refer as the *exclusion of temporal miracles*, for finding inductive formulas that strengthen properties which connect present to past:

If some occurrence in the *past* is going to affect the *future*, it must have some observable trace in the *present*.

This philosophical statement of principle represents both a basic fact about determinateness of programs and a useful approach to the construction of inductive past formulas. The basic fact is that the rest of the computation (or several possible computations) is fully determined by the current state. For anything that happened in the past to influence the rest of the computation, it must have some representation in the current state. For example, if we wish to state that  $x$  can become 10 only after it has already assumed the value 5, we must have some evidence of the fact that  $x = 5$  has already happened. This evidence must be present in all the states that have  $x = 5$  in their past. The obvious choice in this case is  $x > 5$  as characterizing, in terms of the *current* system variables, all the states which have  $x = 5$  in their strict past.

The constructive suggestion implied by the above principle is to search for a state property (represented by a state formula) that is equivalent to, or implies, the past formula. One then constructs a formula claiming some relation (such as implication) between the state formula and the past formula and shows this relation to be a  $P$ -invariant of the program.

### Example (producer-consumer)

Let us illustrate this principle on the producer-consumer program PROD-CONS of Fig. 4.3 (see also Fig. 1.19, page 133). At the same time, this will illustrate a general approach to proofs of properties stating the absence of unsolicited response.

In Section 1.5 (page 133), we proved two important safety properties of this program. First, that channel *send* never contains more than  $N$  messages (values), i.e.,

$$|send| \leq N.$$

The number  $N$  is a parameter of the program, which is assumed to be positive.

The second property states that  $|ack| + |send|$  is essentially equal to  $N$ , except when control is at  $\ell_3$  or at  $m_2$ , where each of these locations independently contributes a deficit of 1 to the sum, i.e.,

$$|ack| + |send| + at_{-\ell_3} + at_{-m_2} = N.$$

An appropriate statement of the fact that no messages are spuriously invented by the transmission system (the *no-invented-messages* property) is given by the

**local** *send, ack: channel [1..] of integer*

**where** *send = Λ, ack =*  $\underbrace{[1, \dots, 1]}_N$

$$\text{Prod} :: \left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \text{ack} \Rightarrow t \\ \ell_3: \text{send} \Leftarrow x \end{array} \right] \end{array} \right] \parallel \text{Cons} :: \left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{send} \Rightarrow y \\ m_2: \text{ack} \Leftarrow 1 \\ m_3: \text{consume } y \end{array} \right] \end{array} \right]$$

Fig. 4.3. Program PROD-CONS (producer-consumer).

invariance of

$$\psi: \text{at\_}m_3 \wedge y = u \rightarrow \Diamond(\text{at\_}\ell_2 \wedge x = u).$$

This formula states that every value assumed by *y*, which process *Cons* is about to consume at *m*<sub>3</sub>, must have been previously produced by *Prod* on moving to *ℓ*<sub>2</sub>. Note the use of the free rigid variable *u* to connect the current value of *y* with a previous value of *x*. This is a typical connection of two remote situations, one at which *u* is about to be consumed, and a much earlier one at which *u* has been freshly produced.

For convenience, we introduce the abbreviations

$$\text{prod}(u): \text{at\_}\ell_2 \wedge x = u$$

$$\text{cons}(u): \text{at\_}m_3 \wedge y = u.$$

With these abbreviations, we will prove the invariance of the formula

$$\psi: \text{cons}(u) \rightarrow \Diamond \text{prod}(u).$$

The principle of exclusion of temporal miracles suggests that if there is a causal link between the appearance of *u* at *ℓ*<sub>2</sub> and its reappearance at *m*<sub>3</sub>, there should be a continuous presence of *u* in all the intermediate states.

Once this is suggested, it is easy to find this continuous presence by tracing the transmigration of *u* from *x* at *ℓ*<sub>2</sub> to *y* at *m*<sub>3</sub>. The proposed inductive formula is:

$$\varphi: \left[ \begin{array}{l} (\text{at\_}\ell_3 \wedge x = u) \vee \\ u \in \text{send} \vee \\ (\text{at\_}m_{2,3} \wedge y = u) \end{array} \right] \rightarrow \Diamond(\text{at\_}\ell_2 \wedge x = u).$$

The antecedent of this formula identifies the possible locations of the value  $u$  during its travel from  $\ell_2$  to  $m_3$ . It can reside in  $x$  while  $Prod$  is at  $\ell_3$ . It then moves to the buffer (list) associated with the channel  $send$ . Finally, it moves to  $y$  while  $Cons$  is at  $m_{2,3}$ . The presence of the value  $u$  in any of these places implies that it originated at  $\ell_2$  sometime in the past.

We shall show by rule INV-P that  $\psi$  is invariant, using the auxiliary past formula  $\varphi$ .

- Premise P1

$$\underbrace{\left[ \dots \vee \dots \vee (at\_m_{2,3} \wedge y = u) \right]}_{\varphi} \rightarrow \Diamond(at\_{\ell_2} \wedge x = u) \Rightarrow$$

$$(at\_m_3 \wedge y = u) \rightarrow \underbrace{\Diamond(at\_{\ell_2} \wedge x = u)}_{\psi} .$$

It is clear that  $\varphi$  entails  $\psi$ .

Let us show that  $\varphi$  is inductive, i.e., satisfies premises P2 and P3.

- Premise P2

$$\underbrace{at\_{\ell_0} \wedge at\_{m_0} \wedge send = \Lambda \wedge \dots}_{\Theta} \rightarrow$$

$$\underbrace{\left[ (at\_{\ell_3} \wedge x = u) \vee u \in send \vee (at\_m_{2,3} \wedge y = u) \right]}_{(\varphi)_0} \rightarrow (at\_{\ell_2} \wedge x = u) .$$

Since  $\Theta$  implies

$$at\_{\ell_0} \wedge at\_{m_0} \wedge send = \Lambda ,$$

it makes the antecedent of  $(\varphi)_0$  false, thus proving the implication.

- Premise P3

For each  $\tau$ , we have to prove

$$\rho_\tau \wedge \underbrace{\left[ (at\_{\ell_3} \wedge x = u) \vee u \in send \vee (at\_m_{2,3} \wedge y = u) \right]}_{\varphi} \rightarrow \Diamond(at\_{\ell_2} \wedge x = u) \Rightarrow$$

$$\left( \underbrace{\left[ \begin{array}{l} (at_{-\ell_3} \wedge x = u) \vee \\ u \in send \vee \\ (at_{-m_{2,3}} \wedge y = u) \end{array} \right]}_{\varphi} \rightarrow \Diamond(at_{-\ell_2} \wedge x = u) \right)'$$

We have to consider only three critical transitions:  $\ell_2$ ,  $\ell_3$ , and  $m_1$ . The reader is requested to complete this proof in **Problem 4.2**.

This concludes the verification of premise P3 for  $\varphi$ , establishing the invariance of  $\psi$ . ■

### The Case of Duplicate Messages

The formula

$$\psi: cons(u) \rightarrow \Diamond prod(u).$$

presented above as a specification of the property of no-invented-messages, does not capture the intuition of this property with full precision. As long as the produced values are pairwise distinct, it provides a satisfactory specification of the desired property. However, once we allow the same value to be produced more than once, thus considering the case of *duplicate messages*, formula  $\psi$  is not completely satisfactory. This is because, while  $\psi$  ensures that every value consumed was previously produced, it also allows a certain value, say 1, to be produced only once and then to be consumed twice, e.g., in the sequence

$$prod(1), \quad cons(1), \quad cons(1), \quad \dots$$

This shows that in the presence of duplicate messages, formula  $\psi$  does not specify the intended property of no-invented-messages with full precision. The fully precise property, to which we refer as the property of *no-invented-replicates*, requires that the  $n$ th consumption of a value be preceded by at least  $n$  productions of this value. In the presence of duplicate messages, this stronger property is not specified by the formula  $\psi$ . Consequently, in the rest of the chapter we assume that the producer in program PROD-CONS never produces the same message twice.

It is a known fact that without using auxiliary variables (equivalently, without quantification), it is impossible to write a temporal formula that will specify the property of no-invented-replicates. In Section 4.7 (page 362), we will show how this property can be specified using auxiliary history variables.

A possible solution to the problem of duplicate messages relies on the principle of *data independence*. A program is called a *data-independent program* if its behavior cannot be influenced by the values of messages or the presence or absence of duplications. Program PROD-CONS of Fig. 4.3 is clearly a data-independent pro-

gram since, besides producing and consuming messages via the schematic statements **produce** and **consume**, the only manipulation of messages by PROD-CONS consists of copying their values from one variable to another, or sending them along channels. In particular, the program never compares the value of one message with the value of another and thus, can never find out if there is a message duplication. We have the following claim about data-independent programs:

### Claim 4.3 (data-independent programs)

If formula  $\psi: \text{cons}(u) \rightarrow \Diamond \text{prod}(u)$  is  $P$ -valid over the data-independent program  $P$ , then the program has the property of no-invented-replicates. That is, for every  $n > 0$  and value  $u$ , the  $n$ th consumption of  $u$  is preceded by at least  $n$  productions of  $u$ .

In **Problem 4.3**, the reader is requested to prove this claim. In **Problem 4.4**, the reader is requested to consider the alternative solution of using quantification in order to specify the property of no-invented-replicates.

## Using Ideal Variables

As was the case with state invariants, the use of ideal variables (i.e., virtual or auxiliary variables) can lead to considerable simplification in proofs of invariances. We illustrate here the use of virtual and auxiliary variables for simplifying the proof of the preceding example.

### The Virtual Channel *send*\*

Consider first the virtual variable *send*\* defined by

$$\text{send}^* = [\text{if } \text{at\_}m_{2,3} \text{ then } (y) \text{ else } \Lambda] * \text{send} * [\text{if } \text{at\_}\ell_3 \text{ then } (x) \text{ else } \Lambda]$$

where '\*' is an operator for concatenating lists.

This definition views  $x$  and  $y$  as extensions of channel *send*. Variable  $y$  extends the front of the channel *send* whenever *Cons* is at  $m_{2,3}$ , while  $x$  extends the back of the channel *send* whenever *Prod* is at  $\ell_3$ .

If we check which transitions modify *send*\*, we find that both  $\ell_3$  and  $m_1$ , which are the writing and reading statements of the real channel *send*, do not modify the virtual *send*\*. This is because *send*\* has been defined as the concatenation of three terms, of which *send* is the middle one, and  $\ell_3$  and  $m_1$  only cause an internal shift of an element from one term to its neighbor. It is precisely our wish to move the modifications of the channel from  $\ell_3$  to  $\ell_2$ , and from  $m_1$  to  $m_3$ , which led to the definition of *send*\* presented above.

The transitions that do modify  $send^*$  are  $\ell_2$  and  $m_3$ . Clearly,  $\ell_2$  adds  $x$  to the end of  $send^*$  (by making  $at_{-\ell_3}$  true), while  $m_3$  removes  $y$  from the front of  $send^*$  (by making  $at_{-m_{2,3}}$  false). Their effect on  $send^*$  can be summarized by saying that transition relation  $\rho_{m_3}$  implies

$$(send^*)' = tl(send^*),$$

and transition relation  $\rho_{\ell_2}$  implies

$$(send^*)' = send^* \bullet x.$$

It is easy to establish the following invariants, relating  $send^*$  to  $x$  and  $y$ :

$$\varphi_1: at_{-\ell_3} \rightarrow x = last(send^*)$$

$$\varphi_2: at_{-m_{2,3}} \rightarrow y = hd(send^*).$$

Here  $last$  refers to the last element in a list.

Using  $send^*$ , we can rewrite formula  $\varphi$  of the previous example as

$$\varphi^*: u \in send^* \rightarrow \Diamond(at_{-\ell_2} \wedge x = u).$$

The verification conditions required by premise P3 for  $\varphi^*$  are proved in a similar way to the proof for  $\varphi$ , except that one must check only two transitions,  $\ell_2$  and  $m_3$ , instead of the three required in the proof of  $\varphi$ .

The proof proceeds to conclude the invariance of

$$\psi: cons(u) \rightarrow \Diamond prod(u).$$

### The Auxiliary Variable $send^a$

An alternative (and closely related) way to introduce an ideal version of channel  $send$  is as an auxiliary variable. In Fig. 4.4 we present program PROD-CONS-A in which  $send^a$  appears as an auxiliary variable.

Note that this program also satisfies the invariant which served as the definition for the virtual variable  $send^*$ :

$$\chi: send^a = [if at_{-m_{2,3}} then (y) else \Lambda] * send^*$$

$$[if at_{-\ell_3} then (x) else \Lambda].$$

As in the case of  $send^*$ , the proof continues by establishing the invariant

$$\varphi^a: u \in send^a \rightarrow \Diamond(at_{-\ell_2} \wedge x = u),$$

from which the invariance of

$$\psi: cons(u) \rightarrow \Diamond prod(u)$$

follows.

```

local      send, ack: channel [1..] of integer
          where send = Λ, ack =  $\underbrace{[1, \dots, 1]}_N$ 
auxiliary senda : list of integer where senda = Λ

Prod :: 
$$\left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{produce } x \\ \ell_2: \langle \text{ack} \Rightarrow t; \text{send}^a := \text{send}^a \bullet x \rangle \\ \ell_3: \text{send} \Leftarrow x \end{array} \right] \end{array} \right]$$

||

Cons :: 
$$\left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{send} \Rightarrow y \\ m_2: \text{ack} \Leftarrow 1 \\ m_3: \langle \text{consume } y \\ \quad \left\langle \text{send}^a := \text{tl}(\text{send}^a) \right\rangle \end{array} \right] \end{array} \right]$$


```

Fig. 4.4. Program PROD-CONS-A (producer-consumer)  
with auxiliary variable.

### 4.3 Compositional Verification

In Section 0.4 (page 36), we introduced the notion of a module and showed how to interpret a module as a fair transition system that takes into account the environment's actions. We also associated a set of modular computations with each top-level process of a program. In page 52, we defined a formula  $\varphi$  to be modularly valid over process  $P_i$ , denoted  $P_i \models_m \varphi$ , if  $\varphi$  holds over all the modular computations of  $P_i$ ; that is, if  $M_i \models \varphi$  for the module  $M_i$  corresponding to process  $P_i$ .

An important observation made earlier is that a computation of a program  $P$  is also a modular computation of each of its top-level processes. This leads to rule MOD presented in Fig. 4.5.

Premise D1 states that  $\chi$  is an invariant of the entire program  $P$ . Premise

For  $P_i$ , a top-level process of program  $P$ , and past formulas  $\chi, p$ ,

$$\frac{\begin{array}{l} \text{D1. } P \models \square \chi \\ \text{D2. } P_i \models_m \square \chi \Rightarrow p \end{array}}{P \models \square p}$$

Fig. 4.5. Rule MOD (compositional verification).

D2 states that the entailment  $\square \chi \Rightarrow p$  is modularly valid over some top-level process  $P_i$ . From these two assumptions, the rule infers that  $p$  is an invariant of  $P$ .

**Justification** Assume that premises D1 and D2 hold and let  $\sigma$  be a computation of program  $P$ . By premise D1, formula  $\chi$  holds at all positions of  $\sigma$ . Since every computation of  $P$  is also a modular computation of  $P_i$ , premise D2 implies that the formula  $\square \chi \rightarrow p$  holds at all positions of  $\sigma$ .

Consider an arbitrary position  $j \geq 0$  of  $\sigma$ . By D1,  $\chi$  holds at all positions  $k \leq j$  and, therefore  $\square \chi$  holds at  $j$ . By D2,  $\square \chi \rightarrow p$  holds at  $j$  and, therefore so does  $p$ .

We conclude that  $p$  holds at all positions of  $\sigma$ . ■

Rule MOD is often used in an incremental style. As a first step we take  $\chi = \top$  and prove  $P_i \models_m \square p_1$ . From this the rule infers

$$P \models \square p_1.$$

Next, we take  $\chi = p_1$  and prove  $P_i \models_m \square p_1 \Rightarrow p_2$ . This leads to

$$P \models \square p_2,$$

which may be followed by additional steps.

The advantage of this proof pattern is that in each step we concentrate on proving modular validity over a single process  $P_i$ . If  $P_i$  is only a small part of the program, each compositional verification step has to consider only a small fraction of the transitions in the complete program.

We illustrate the use of rule MOD on a simple example.

### Example (program KEEPING-UP)

Consider program KEEPING-UP presented in Fig. 4.6 (see also Fig. 0.12, page 40). Process  $P_1$  in this program repeatedly increments  $x$ , provided  $x$  does not exceed

**local**  $x, y$ : integer **where**  $x = y = 0$

$$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{await } x < y + 1 \\ \ell_2: x := x + 1 \end{array} \right] \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{await } y < x + 1 \\ m_2: y := y + 1 \end{array} \right] \end{array} \right]$$

Fig. 4.6. Program KEEPING-UP.

$y + 1$ . In a symmetric way, process  $P_2$  repeatedly increments  $y$ , provided  $y$  does not exceed  $x + 1$ .

We wish to prove for this program the invariance of the assertion  $|x - y| \leq 1$ , i.e.,

$$\square \underbrace{|x - y| \leq 1}_{p},$$

claiming that the difference between  $x$  and  $y$  never exceeds 1 in absolute value.

We prove this property by compositional verification, using rules INV-P and MOD. We first show the  $P$ -validities

$$P \models \square(x \geq x^-) \quad \text{and} \quad P \models \square(y \geq y^-),$$

and then the  $P$ -validities

$$P \models \square(x \leq y + 1) \quad \text{and} \quad P \models \square(y \leq x + 1).$$

The invariants  $x \leq y + 1$  and  $y \leq x + 1$  imply the desired  $P$ -validity

$$P \models \square(|x - y| \leq 1).$$

In Fig. 4.7 and Fig. 4.8, we present processes  $P_1$  and  $P_2$  of program KEEPING-UP as modules. The programs for these modules appeared also in Fig. 0.13 (page 40) and Fig. 0.14 (page 41).

*Proof of  $P \models \square(x \geq x^-)$  and  $P \models \square(y \geq y^-)$*

First, we prove the modular invariance of the formula

$$\varphi_1: x \geq x^-,$$

claiming that  $x$  never decreases, over process  $P_1$ . This means that  $\varphi_1$  holds at all positions of every modular computation of process  $P_1$ . To do so, we prove

$$M_1 \models \square \underbrace{(x \geq x^-)}_{\varphi_1}$$

```

module M1
in y : integer where y = 0
own out x : integer where x = 0

l0: loop forever do
     $\left[ \begin{array}{l} \ell_1: \text{await } x < y + 1 \\ \ell_2: x := x + 1 \end{array} \right]$ 

```

Fig. 4.7. Module *M*<sub>1</sub> corresponding to process *P*<sub>1</sub>.

```

module M2
in x : integer where x = 0
own out y : integer where y = 0

m0: loop forever do
     $\left[ \begin{array}{l} m_1: \text{await } y < x + 1 \\ m_2: y := y + 1 \end{array} \right]$ 

```

Fig. 4.8. Module *M*<sub>2</sub> corresponding to process *P*<sub>2</sub>.

for module *M*<sub>1</sub> corresponding to process *P*<sub>1</sub>, using rule INV-P with  $p = \varphi = \varphi_1$ .

- Premise P1 obviously holds, since  $p = \varphi$ .
- Premise P2 is trivial since  $(x \geq x^-)_0$  simplifies to  $x \geq x$ .
- Premise P3 has to be checked for process transitions  $\ell_0$ ,  $\ell_1$ , and  $\ell_2$ , for the idling transition  $\tau_I$ , and the environment transition  $\tau_E$ . Since *x* is an *own output* variable for *P*<sub>1</sub>,  $\rho_{\tau_E}$  implies  $x' = x$  from which  $(x \geq x^-)' = x' \geq x$  follows. As  $\tau_I$ ,  $\ell_0$ , and  $\ell_1$  do not modify *x*, their transition relations also imply  $\varphi'_1$ . It remains to check P3 for  $\ell_2$ , leading to the valid verification condition

$$\underbrace{\dots \wedge x' = x + 1}_{\rho_{\ell_2}} \wedge \underbrace{\dots}_{\varphi_1} \Rightarrow \underbrace{x' \geq x}_{\varphi'_1}.$$

We conclude

$$P_1 \models_m \Box(x \geq x^-).$$

We now use rule MOD with  $\chi: \tau$  and  $p: x \geq x^-$ , to conclude

$$P \models \square(x \geq x^-).$$

In a symmetric way, we can establish

$$P_2 \models_m \square(y \geq y^-),$$

leading to

$$P \models \square(y \geq y^-).$$

*Proof of  $P \models \square(x \leq y + 1)$  and  $P \models \square(y \leq x + 1)$*

As the next step, we prove the modular invariance of the past formula

$$M_1 \models \underbrace{\square(y \geq y^-)}_{\psi_1} \rightarrow x \leq y + 1.$$

This formula states that if up to now,  $y$  has never decreased, then  $x$  does not exceed  $y + 1$ .

To prove this, we use rule INV-P with  $p = \psi_1$  and

$$\varphi: \square(y \geq y^-) \rightarrow x \leq y + at_{-}\ell_{0,1}.$$

To simplify the presentation of the proof we rewrite  $p$  and  $\varphi$  in the following equivalent forms:

$$p: \Diamond(y < y^-) \vee x \leq y + 1$$

$$\varphi: \Diamond(y < y^-) \vee x \leq y + at_{-}\ell_{0,1}.$$

Assertion  $p$  states that either  $y$  was observed to decrease in the past or  $x$  does not exceed  $y + 1$ .

We consider each premise of rule INV-P.

- Premise P1 requires showing the following entailment

$$\underbrace{\left( \begin{array}{c} \Diamond(y < y^-) \\ \vee \\ x \leq y + at_{-}\ell_{0,1} \end{array} \right)}_{\varphi} \Rightarrow \underbrace{\left( \begin{array}{c} \Diamond(y < y^-) \\ \vee \\ x \leq y + 1 \end{array} \right)}_{p}.$$

Since  $at_{-}\ell_{0,1} \leq 1$ , the entailment is valid.

- Premise P2 reduces to the following implication

$$\underbrace{\cdots \wedge x = y = 0}_{\Theta} \rightarrow \underbrace{y < y \vee x \leq y + at_{-}\ell_{0,1}}_{(\varphi)_0}.$$

The implication is valid since  $at_{-}\ell_{0,1} \geq 0$ .

- Premise P3 should be checked for transitions  $\ell_0$ ,  $\ell_1$ ,  $\ell_2$ ,  $\tau_I$ , and  $\tau_E$ . To establish the verification condition

$$\rho_T \wedge \underbrace{\left( \begin{array}{c} \Diamond(y < y^-) \\ \vee \\ x \leq y + at_{-}\ell_{0,1} \end{array} \right)}_{\varphi} \Rightarrow \underbrace{\left( \begin{array}{c} \Diamond(y < y^-) \vee y' < y \\ \vee \\ x' \leq y' + at'_{-}\ell_{0,1} \end{array} \right)}_{\varphi'}$$

it suffices to prove the implication

$$\rho_T \wedge x \leq y + at_{-}\ell_{0,1} \rightarrow y' < y \vee x' \leq y' + at'_{-}\ell_{0,1},$$

which is equivalent to

$$\rho_T \wedge x \leq y + at_{-}\ell_{0,1} \wedge y \leq y' \rightarrow x' \leq y' + at'_{-}\ell_{0,1}.$$

Let us consider each of the transitions.

- *Transition  $\ell_0$*  — Since  $\rho_{\ell_0}$  implies  $x' = x$ ,  $y' = y$ , and  $at'_{-}\ell_{0,1} = at_{-}\ell_{0,1}$ , the implication is valid.
- *Transition  $\ell_1$*  — Transition relation  $\rho_{\ell_1}$  implies  $x < y+1$ ,  $x' = x$ , and  $y' = y$ . It follows that  $x' \leq y' \leq y' + at'_{-}\ell_{0,1}$ .
- *Transition  $\ell_2$*  — Transition relation  $\rho_{\ell_2}$  implies  $x' = x+1$ ,  $y' = y$ ,  $at_{-}\ell_{0,1} = 0$ , and  $at'_{-}\ell_{0,1} = 1$ . We can compute

$$y' + at'_{-}\ell_{0,1} - x' = y+1 - (x+1) = y - x = y + at_{-}\ell_{0,1} - x \geq 0.$$

- *Transitions  $\tau_I$ ,  $\tau_E$*  — The transition relations for both transitions imply  $x' = x$  and  $at'_{-}\ell_{0,1} = at_{-}\ell_{0,1}$ . Using the clause  $y \leq y'$ , we can compute

$$y' + at'_{-}\ell_{0,1} - x' = y' + at_{-}\ell_{0,1} - x \geq y + at_{-}\ell_{0,1} - x \geq 0.$$

We conclude

$$P_1 \models_m \square \left( \underbrace{\Box(y \geq y^-) \rightarrow x \leq y+1}_{\psi_1} \right),$$

which can also be written as

$$P_1 \models_m \Box(y \geq y^-) \Rightarrow x \leq y+1.$$

We use rule MOD with  $\chi: y \geq y^-$  and  $p: x \leq y+1$  to conclude

$$P \models \square(x \leq y+1).$$

In a symmetric way, we can establish

$$P_2 \models_m \square(\Box(x \geq x^-) \rightarrow y \leq x+1),$$

and use rule MOD to conclude

$$P \models \square(y \leq x+1).$$

The invariants  $x \leq y+1$  and  $y \leq x+1$  imply the validity

$$P \models \square(|x-y| \leq 1).$$

In **Problem 4.5**, the reader is requested to prove the invariance of the assertion  $|x - y| \leq 1$  in a non-compositional style, using the methods of Chapters 1 and 2.

## 4.4 Causality Rule

In Section 4.2 we showed how causality properties, i.e., properties expressible by formulas of the form  $p \Rightarrow \Diamond r$  for past formulas  $p$  and  $r$ , can be proved using the general rule INV-P. In this section we introduce a rule that is specially tailored for proving causality properties.

### The Inverse Verification Condition

To prove causality properties we need to reason *backwards*. The formula  $p \Rightarrow \Diamond r$  states that any  $p$ -state must be preceded by an  $r$ -state. In general, the two events  $p$  and  $r$  may be quite remote in time. Consistent with our general approach to proving temporal properties, we try to reduce such a property to a local case of causality. The local case is when the immediate predecessor of any  $p$ -state is an  $r$ -state.

This leads us to the consideration of the *inverse verification condition*

$$\{\varphi\} \tau^{-1} \{\psi\}: \rho_\tau \wedge \varphi' \Rightarrow \psi.$$

For  $\varphi$  and  $\psi$  being state formulas, this entailment claims that

if the next state, achieved by performing  $\tau$ , satisfies  $\varphi$ ,  
then the current state must satisfy  $\psi$ .

For the case that  $\varphi$  and  $\psi$  are past formulas, a similar relation holds between positions  $i$  and  $i + 1$  such that  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ .

The consequences of having a  $P$ -valid inverse verification condition are stated by the following proposition:

#### Proposition 4.4 (inverse verification condition)

Let  $\{\varphi\} \tau^{-1} \{\psi\}$  be  $P$ -valid and  $\sigma$  be a  $P$ -computation. If  $(\sigma, i + 1) \models \varphi$  and  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ , then  $(\sigma, i) \models \psi$ .

**Justification** Let  $s_i$  and  $s_{i+1}$  be two consecutive states in a computation  $\sigma$  of  $P$ . Assume further, that  $s_{i+1}$  is a  $\tau$ -successor of  $s_i$ , and that  $(\sigma, i + 1) \models \varphi$ . We have to show that  $(\sigma, i) \models \psi$ .

Let  $\tilde{\sigma}: \tilde{s}_0, \dots, \tilde{s}_i, \tilde{s}_{i+1}, \dots$  be the predictive version of  $\sigma$ . It is obvious that the inverse verification condition  $\rho_\tau \wedge \varphi' \Rightarrow \psi$  holds at position  $i$  of  $\tilde{\sigma}$ ,  $\tilde{s}_{i+1}$

is a  $\tau$ -successor of  $\tilde{s}_i$ , and  $(\tilde{\sigma}, i+1) \models \varphi$ . It follows that  $(\tilde{\sigma}, i) \models \rho_\tau \wedge \varphi'$  and, by the verification condition,  $(\tilde{\sigma}, i) \models \psi$ . The same must be true of the original computation  $\sigma$ , and we conclude  $(\sigma, i) \models \psi$ . ■

It is interesting to relate the regular verification condition to the inverse condition. The relation between them is given by

$$\{\varphi\} \tau^{-1} \{\psi\} \quad \text{iff} \quad \{\neg\psi\} \tau \{\neg\varphi\}.$$

This can be explained in terms of the entailments corresponding to these verification conditions:

$$\rho_\tau \wedge \varphi' \Rightarrow \psi \quad \text{and} \quad \rho_\tau \wedge \neg\psi \Rightarrow \neg\varphi'.$$

As before, we can extend the definition to cover a set of transitions. Thus, for a set of transitions  $T \subseteq \mathcal{T}$  and the entire set of transitions  $\mathcal{T}$ , we write  $\{\varphi\} T^{-1} \{\psi\}$  and  $\{\varphi\} \mathcal{T}^{-1} \{\psi\}$  to denote the fact that  $\{\varphi\} \tau^{-1} \{\psi\}$  holds for every  $\tau \in T$  and every  $\tau \in \mathcal{T}$ , respectively.

## The CAUS Rule

While causality properties can be handled, as previously, by the general rule INV-P, it is worthwhile to develop a specialized form of rule INV-P that is tailored for causality properties. This customized rule is rule CAUS (Fig. 4.9).

For past formulas  $p, r, \varphi$

$$\text{C1. } p \Rightarrow \varphi \vee r$$

$$\text{C2. } \Theta \rightarrow \neg(\varphi)_0 \vee (r)_0$$

$$\text{C3. } \{\varphi\} \mathcal{T}^{-1} \{\varphi \vee r\}$$

$$\frac{}{p \Rightarrow \Diamond r}$$

Fig. 4.9. Rule CAUS (causality).

Premise C1 of the rule states that any  $p$ -position must either be an  $r$ -position (establishing the conclusion immediately), or satisfy an intermediate assertion  $\varphi$ , bridging between the present state and the preceding state in which  $r$  holds. Premise C3, which is the inverse verification condition  $\{\varphi\} \tau^{-1} \{\varphi \vee r\}$ , for every  $\tau \in \mathcal{T}$ , states that the immediate predecessor of a  $\varphi$ -position must satisfy either  $r$  or  $\varphi$  again. By C1 and C3,  $\varphi$  must propagate backwards from any  $p$ -position to a position satisfying  $r$ , or perhaps reach all the way to the beginning of the computation. Premise C2 disallows this last possibility by stating that any position

satisfying  $\Theta$  (as all initial positions do) cannot satisfy  $(\varphi)_0$ , the initial version of  $\varphi$ , unless it also satisfies  $(r)_0$ .

Note the similarity between premises C1 and C3, which ensure the backward propagation of  $\varphi$  from  $p$  to  $r$ , and premises W1 and W3 of the WAIT rule (Fig. 3.3), which ensure the forward propagation of  $\varphi$  from  $p$  to  $r$ .

For the case of a simple causality property, where  $p$  and  $r$  are state formulas, it is sufficient to use state formulas for  $\varphi$ .

### Rule CAUS as a Special Case of Rule INV-P

Rule CAUS can be formally derived from rule INV-P.

Assume we have formulas  $q$ ,  $r$ , and  $\chi$  satisfying premises C1–C3 of rule CAUS, where we replace formulas  $p$  and  $\varphi$  appearing in the premises by  $q$  and  $\chi$ , to avoid name conflicts with identically named formulas occurring in rule INV-P. This means that the following premises hold:

$$C1. \quad q \Rightarrow \chi \vee r$$

$$C2. \quad \Theta \rightarrow \neg(\chi)_0 \vee (r)_0$$

$$C3. \quad \{\chi\} \tau^{-1} \{\chi \vee r\} \text{ for every } \tau \in T,$$

and we wish to establish the  $P$ -validity of the consequence:

$$q \Rightarrow \Diamond r.$$

First, we establish, using rule INV-P, that  $p: \chi \rightarrow \Diamond r$  is an invariant:

$$\Box p: \chi \Rightarrow \Diamond r$$

is  $P$ -valid. Then, using C1, we conclude

$$q \Rightarrow \Diamond r \vee r,$$

which is equivalent, by the basic properties of the operator  $\Diamond$ , to

$$q \Rightarrow \Diamond r.$$

To show the invariance of  $p$  by rule INV-P, we take

$$\varphi = p: \chi \rightarrow \Diamond r.$$

- Premise P1 of rule INV-P,  $\varphi \Rightarrow p$ , is trivially satisfied.
- Premise P2 reads:

$$\Theta \rightarrow \left( \underbrace{\chi \rightarrow \Diamond r}_{\varphi} \right)_0.$$

By the definition  $(\Diamond r)_0 = (r)_0$ , the implication can be simplified to

$$\Theta \rightarrow ((\chi)_0 \rightarrow (r)_0)$$

which is premise C2.

- Premise P3 reads:

$$\rho_T \wedge \underbrace{x \rightarrow \Diamond r}_{\varphi} \Rightarrow \underbrace{x' \rightarrow (\Diamond r)'}_{\varphi'}.$$

By the definition of the primed version of past temporal formulas

$$(\Diamond r)' = r' \vee \Diamond r.$$

We thus obtain

$$\rho_T \wedge (x \rightarrow \Diamond r) \Rightarrow (x' \rightarrow r' \vee \Diamond r).$$

To prove this it is sufficient to show (by propositional reasoning)

$$\rho_T \wedge (x \rightarrow \Diamond r) \wedge x' \Rightarrow \Diamond r.$$

Since by C3,  $\rho_T \wedge x'$  entails  $x \vee r$  which entails  $x \vee \Diamond r$ , it follows that the left-hand side  $\rho_T \wedge (x \rightarrow \Diamond r) \wedge x'$  entails  $\Diamond r$ .

Thus, by rule INV-P it follows that  $\Box p$  is  $P$ -valid.

In Problem 4.6, the reader is requested to consider a variant of rule CAUS and compare the proving power of the two rules.

To illustrate the use of rule CAUS, we start with a trivial example.

**Example** Let us use rule CAUS to reprove the property

$$\underbrace{x=10}_p \Rightarrow \Diamond \underbrace{x=5}_r,$$

for system INC presented in Fig. 4.2.

Intending to use rule CAUS, it is necessary to find an intermediate assertion  $\varphi$  connecting these two events.

The strategy for constructing  $\varphi$  is the one we described before. That is, find within the current state evidence of the fact that  $x = 5$  has happened before. Formally, we may be helped by the premises of rule CAUS, which suggest that  $\varphi$  should be implied by  $p \wedge \neg r$  and propagate backwards until stopped by  $r$ . With these suggestions, it is not difficult to come up with the assertion

$$\varphi: x > 5.$$

The three premises we have to check are given, following some simplifications, by

$$C1. \quad \underbrace{x=10}_p \Rightarrow \underbrace{x>5}_{\varphi} \vee \underbrace{\dots}_r$$

|

$$\text{C2. } \underbrace{x = 0 \wedge \dots}_{\Theta} \rightarrow \underbrace{x \leq 5}_{\neg(\varphi)_0} \vee \underbrace{\dots}_{(r)_0}$$

$$\text{C3. } \underbrace{x' = x + 1 \wedge \dots}_{p_r} \wedge \underbrace{x' > 5}_{\varphi'} \Rightarrow \underbrace{x \geq 5}_{\varphi \vee r} .$$

All three formulas are obviously valid.

It therefore follows by rule CAUS that  $x = 10 \Rightarrow \Diamond(x = 5)$ , as we wanted to show. ■

In Problem 4.7, the reader is requested to consider alternative formulas for expressing and verifying causality properties.

### Example (producer-consumer)

As a less trivial example of the use of rule CAUS, let us reconsider the property of program PROD-CONS (Fig. 4.3), stating that the system delivers to the consumer only messages that were created by the producer (see Section 4.2). Under our standing assumption of no duplicate messages, this property can be expressed by the formula

$$\underbrace{at\_m_3 \wedge y = u}_p \Rightarrow \Diamond \underbrace{at\_{\ell_2} \wedge x = u}_r .$$

As before, to apply rule CAUS, we have to construct an intermediate assertion  $\varphi$  that tells us where the value  $u$  resides at all intermediate states between its production which is concluded at  $\ell_2$  and its consumption at  $m_3$ .

A reasonable candidate for  $\varphi$  is:

$$\varphi: (at\_{\ell_3} \wedge x = u) \vee u \in send \vee (at\_{m_{2,3}} \wedge y = u).$$

It is easy to express the assertion  $\varphi$  in terms of the virtual variable  $send^*$ , which has been defined as

$$send^* = [if at\_{m_{2,3}} \text{ then } (y) \text{ else } \Lambda] * send * [if at\_{\ell_3} \text{ then } (x) \text{ else } \Lambda].$$

Using this variable,  $\varphi$  is simply expressed as:

$$\varphi: u \in send^*.$$

The first two premises that have to be checked can be simplified to:

$$\text{C1. } \underbrace{at\_m_3 \wedge y = u}_p \Rightarrow \underbrace{u \in send^*}_{\varphi} \vee \underbrace{\dots}_r .$$

The validity of C1 follows from the invariant

$$\varphi_2: at\_{m_{2,3}} \rightarrow y = hd(send^*),$$

implied by the definition of  $send^*$ .

$$\text{C2. } \underbrace{\at{\ell_0} \wedge \at{m_0} \wedge \text{send} = \Lambda \wedge \dots}_{\Theta} \rightarrow \underbrace{u \notin \text{send}^*}_{\neg(\varphi)_0} \vee \underbrace{\dots}_{(r)_0}.$$

Since  $\Theta$  implies  $\neg at_{m2,3}$ ,  $\text{send} = \Lambda$ , and  $\neg at_{\ell_3}$ , premise C2 is obviously valid.

To check C3, we have to consider only the critical transitions which modify  $\text{send}^*$ . As discussed above, these transitions are  $\ell_2$  and  $m_3$ .

- Transition  $\ell_2$

The inverse verification condition for  $\ell_2$  is:

$$\rho_{\ell_2} \wedge \underbrace{u \in (\text{send}^*)'}_{\varphi'} \Rightarrow \underbrace{u \in \text{send}^*}_{\varphi} \vee \underbrace{\at{\ell_2} \wedge x = u}_{r}.$$

Obviously,  $\rho_{\ell_2}$  implies  $(\text{send}^*)' = \text{send}^* \bullet x$ . Therefore,  $u \in (\text{send}^*)'$  implies that either  $u \in \text{send}^*$  or  $x = u$ . The case  $u \in \text{send}^*$  satisfies the first disjunct on the right-hand side of the verification condition. Since  $\rho_{\ell_2}$  implies  $\at{\ell_2}$ , the case  $x = u$  satisfies the second disjunct in the verification condition.

It follows that the verification condition is valid for the transition  $\ell_2$ .

- Transition  $m_3$

The inverse verification condition for  $m_3$  is:

$$\rho_{m_3} \wedge \underbrace{u \in (\text{send}^*)'}_{\varphi'} \Rightarrow \underbrace{u \in \text{send}^*}_{\varphi} \vee \underbrace{\at{\ell_2} \wedge x = u}_{r}.$$

Since  $\rho_{m_3}$  implies  $(\text{send}^*)' = tl(\text{send}^*)$ ,  $u \in (\text{send}^*)'$  implies  $u \in \text{send}^*$ . ■

## 4.5 Backward Analysis

Causality formulas can be used very effectively, not only for proving causality properties, but also for proving other safety properties. The most striking example of such an application is the proof by contradiction of simple invariances.

Assume that we wish to prove a property that can be expressed as  $\square \neg \psi$ , i.e., claiming that  $\psi$  never happens. The proof begins by assuming that  $\psi$  actually happens. For example, if we wish to prove mutual exclusion of two processes,  $\psi$  may describe a situation in which both processes are at their critical sections at the same time. The approach is viable for the case that  $\psi$  is a state formula, or more generally, a past formula.

We reason backwards, using rule CAUS to establish a chain of assertions  $\psi = \psi_1, \psi_2, \dots$  such that  $\psi_1$  must be preceded by  $\psi_2$ , which must be preceded by  $\psi_3$ , and so on. Contradiction is obtained by reaching an assertion  $\psi_k$  which cannot be satisfied by any  $P$ -accessible state. This shows that  $\psi$  can never arise, and

hence  $\neg\psi$  is an invariant of the program.

## Inference Rules

To formally support such backward reasoning, we need several inference rules that suffice for reasoning about causality formulas. In all these rules,  $p$ ,  $q$ , and  $r$  stand for past formulas.

The first inference rule claims that causality is transitive.

Rule TRN-C (transitivity of causality)

For past formulas  $p$ ,  $q$ , and  $r$ ,

$$\frac{p \Rightarrow \Diamond q \quad q \Rightarrow \Diamond r}{p \Rightarrow \Diamond r}$$

This rule states that if every  $p$  is preceded by a  $q$ , and every  $q$  is preceded by an  $r$ , then every  $p$  must be preceded by an  $r$ .

The second rule provides case-splitting capability.

Rule SPL-C (case splitting for causality)

For past formulas  $p$ ,  $q$ , and  $r$ ,

$$\frac{p \Rightarrow \Diamond r \quad q \Rightarrow \Diamond r}{p \vee q \Rightarrow \Diamond r}$$

This rule states that if both  $p$  and  $q$  must be preceded by an  $r$ , then a situation in which one (or both) of  $p$  or  $q$  holds (but we may not know which), must also be preceded by an  $r$ .

The next rule describes the monotonicity properties of causality formulas.

Rule MON-C (monotonicity of causality)

For past formulas  $p$ ,  $q$ ,  $r$ , and  $u$ ,

$$\frac{p \Rightarrow q \quad q \Rightarrow \Diamond r \quad r \Rightarrow u}{p \Rightarrow \Diamond u}$$

This rule states that if  $q \Rightarrow \Diamond r$  is valid, we may replace  $q$  by a stronger formula  $p$ , and  $r$  by a weaker formula  $u$  to obtain a valid causality.

Finally, we present the rule that is used to derive the conclusion from an obtained contradiction.

Rule CONT (contradiction)

For a past formula  $p$ ,

$$\frac{p \Rightarrow \Diamond F}{\Box \neg p}$$

This rule states that, if we have established that every  $p$ -state is preceded by an impossible situation, then  $p$  itself is impossible, and hence can never appear in a computation of the program under study.

## Backward Analysis Proof

Let us demonstrate the technique.

**Example** (Peterson's algorithm — version 2)

Consider program MUX-PET2 of Fig. 4.10 (see Fig. 1.14, page 124, and Fig. 3.7, page 268), implementing mutual exclusion by shared variables.

The safety property of mutual exclusion is stated by the invariance

$$\Box \neg \left( \underbrace{at\_l_5 \wedge at\_m_5}_{\psi} \right).$$

We apply backward reasoning to show that the situation described by  $\psi$  is impossible.

In all the proofs we freely use the simple invariances that were already derived for program MUX-PET2 (page 124):

$$\chi_0: s = 1 \vee s = 2$$

$$\chi_1: y_1 \leftrightarrow at\_l_{3..6}$$

$$\chi_2: y_2 \leftrightarrow at\_m_{3..6}.$$

First we establish

$$\psi \Rightarrow \Diamond \left( \underbrace{at\_l_4 \wedge at\_m_5 \wedge s = 2}_{\psi_1} \vee \underbrace{at\_l_5 \wedge at\_m_4 \wedge s = 1}_{\psi_2} \right).$$

```
local y1, y2: boolean where y1 = F, y2 = F
      s      : integer where s = 1
```

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: y_1 := T \\ \ell_3: s := 1 \\ \ell_4: \left[ \begin{array}{l} \ell_4^a: \text{await } \neg y_2 \\ \text{or} \\ \ell_4^b: \text{await } s \neq 1 \end{array} \right] \\ \ell_5: \text{critical} \\ \ell_6: y_1 := F \end{array} \right] \end{array} \right]$

||

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: y_2 := T \\ m_3: s := 2 \\ m_4: \left[ \begin{array}{l} m_4^a: \text{await } \neg y_1 \\ \text{or} \\ m_4^b: \text{await } s \neq 2 \end{array} \right] \\ m_5: \text{critical} \\ m_6: y_2 := F \end{array} \right] \end{array} \right]$

Fig. 4.10. Program MUX-PET2 (Peterson's algorithm) — refined version.

- Establishing  $\psi \Rightarrow \Diamond(\psi_1 \vee \psi_2)$

To prove  $\psi \Rightarrow \Diamond(\psi_1 \vee \psi_2)$ , we use rule CAUS with

$$p = \varphi: \quad \psi \quad \text{and} \quad r: \quad \psi_1 \vee \psi_2.$$

This choice is suggested by the fact that we expect any  $\psi$ -state to be immediately preceded by a state satisfying  $\psi_1 \vee \psi_2$ . Hence  $\varphi$  can coincide with  $\psi$ .

Review the first two premises that have to be verified:

$$\text{C1. } \underbrace{\psi}_{p} \Rightarrow \underbrace{\psi}_{\varphi} \vee \underbrace{\dots}_{r}$$

$$\text{C2. } \underbrace{at_{-\ell_0} \wedge at_{-m_0} \wedge \dots}_{\Theta} \rightarrow \neg \left( \underbrace{at_{-\ell_5} \wedge at_{-m_5} \wedge \dots}_{(\varphi)_0} \right) \vee \underbrace{\dots}_{(r)_0}.$$

Premises C1 and C2 trivially hold.

Next, consider C3. We have to consider only transitions  $\tau$  under which it is possible that  $\varphi'$  holds but  $\varphi$  does not. These are easily identified as  $\ell_4^a$ ,  $\ell_4^b$ ,  $m_4^a$ , and  $m_4^b$ .

- Transition  $\ell_4^a$

For the inverse verification condition of  $\ell_4^a$ , we prove

$$\underbrace{at_{-\ell_4} \wedge at'_{-m_5} = at_{-m_5} \wedge \neg y_2 \wedge \dots}_{\rho_{\ell_4^a}} \wedge \underbrace{at'_{-\ell_5} \wedge at'_{-m_5}}_{\varphi'} \rightarrow \underbrace{\dots}_{\varphi} \vee \underbrace{\dots}_{r}.$$

Since transition relation  $\rho_{\ell_4^a}$  implies  $at'_{-m_5} = at_{-m_5}$  and  $\varphi'$  implies  $at'_{-m_5} = \top$ , we conclude that  $at_{-m_5} = \top$ . By  $\chi_2$ , this implies that  $y_2 = \top$ . Thus the left-hand side of the implication is false.

The case of transition  $m_4^a$  is handled similarly, showing that  $\neg y_1 = \text{F}$ .

- Transition  $\ell_4^b$

For the inverse verification condition  $\ell_4^b$ , we prove

$$\underbrace{at_{-\ell_4} \wedge at'_{-m_5} = at_{-m_5} \wedge s \neq 1 \wedge \dots}_{\rho_{\ell_4^b}} \wedge \underbrace{at'_{-\ell_5} \wedge at'_{-m_5}}_{\varphi'} \rightarrow \underbrace{\dots}_{\varphi} \vee \underbrace{\at_{-\ell_4} \wedge at_{-m_5} \wedge s = 2}_{\psi_1} \stackrel{\tau}{\wedge} \underbrace{\dots}_{\psi_2}.$$

As before,  $\rho_{\ell_4^b}$  and  $\varphi'$  implies that  $at_{-m_5} = \top$ . By  $\chi_0$ ,  $s \neq 1$  implies that  $s = 2$ . Thus  $\psi_1$  holds.

The case of transition  $m_4^b$  can be handled similarly, establishing  $\psi_2$ .

This establishes  $\psi \Rightarrow \Diamond(\psi_1 \vee \psi_2)$ , which states that every  $\psi$ -state must be preceded by a  $(\psi_1 \vee \psi_2)$ -state. We now continue tracing the history leading to a  $\psi_1$ -state. A similar analysis can be applied to  $\psi_2$ -states.

An examination of  $\psi_1$ , whose control part is  $at_{-\ell_4} \wedge at_{-m_5}$ , suggests two candidates for predecessors. They are  $at_{-\ell_3} \wedge at_{-m_5}$  and  $at_{-\ell_4} \wedge at_{-m_4}$ . However,

a more careful analysis shows that neither of them can really be a predecessor of  $\psi_1$ .

Stated informally,  $at_{-\ell_3} \wedge at_{-m_5}$  cannot be a predecessor of a  $\psi_1$ -state because the transition  $\ell_3$ , which is expected to lead to  $\psi_1$ , sets  $s$  to 1 instead of the value 2 which  $\psi_1$  requires. Neither can the situation  $at_{-\ell_4} \wedge at_{-m_4}$  be a predecessor of a  $\psi_1$ -state, since transitions  $m_4^a$  and  $m_4^b$ , which are the only ones that connect this situation to  $\psi_1$ , are enabled only when  $y_1 = F$  or when  $s = 1$ . We will show that none of these conditions can hold on a state which is an  $m_4^a$ -predecessor or an  $m_4^b$ -predecessor of a  $\psi_1$ -state. This suggests that  $\psi_1$  can be proven to be an impossible situation.

Consequently, we will prove:

$$\underbrace{at_{-\ell_4} \wedge at_{-m_5} \wedge s = 2}_{\psi_1} \Rightarrow \Diamond F.$$

- Establishing  $\psi_1 \Rightarrow \Diamond F$

We use rule CAUS, taking

$$p = \varphi: \psi_1 \quad \text{and} \quad r: F.$$

Consider the three premises of rule CAUS.

Premise C1,

$$\underbrace{\psi_1}_p \Rightarrow \underbrace{\psi_1}_{\varphi} \vee \underbrace{\dots}_r,$$

is trivially valid.

Premise C2,

$$\underbrace{at_{-\ell_0} \wedge at_{-m_0} \wedge \dots}_{\Theta} \rightarrow \underbrace{\neg(at_{-\ell_4} \wedge at_{-m_5} \wedge s = 2)}_{\neg(\varphi)_0} \vee \underbrace{\dots}_{(r)_0},$$

is clearly valid.

Premise C3 has to be checked over all transitions  $\tau$  that allow a true  $\psi'_1$  and a false  $\psi_1$ . The only serious candidates are  $\ell_3$ ,  $m_4^a$ , and  $m_4^b$ . Let us consider each in turn.

- Transition  $\ell_3$

The inverse verification condition for  $\ell_3$  is

$$\underbrace{\dots \wedge s' = 1}_{p_{\ell_3}} \wedge \underbrace{\dots \wedge s' = 2}_{\varphi'} \rightarrow \underbrace{\dots}_{\varphi \vee \tau}.$$

The left-hand side of the implication is false, and therefore the implication is valid.

- Transition  $m_4^a$

The inverse verification condition for  $m_4^a$  is

$$\underbrace{\dots \wedge \neg y_1 \wedge at'_\ell \ell_4 = at_\ell \ell_4}_{\rho_{m_4^a}} \wedge \underbrace{at'_\ell \ell_4 \wedge \dots}_{\varphi'} \rightarrow \underbrace{\dots}_{\varphi \vee \tau}.$$

Observing that  $\rho_{m_4^a}$  implies  $\neg y_1$ , and that by invariant  $\chi_1$ ,  $at_\ell \ell_4$  implies  $y_1$ , it follows that the left-hand side of the implication is false.

- Transition  $m_4^b$

The inverse verification condition for  $m_4^b$  is

$$\underbrace{\dots \wedge s \neq 2 \wedge s' = s}_{\rho_{m_4^b}} \wedge \underbrace{\dots \wedge s' = 2}_{\varphi'} \rightarrow \underbrace{\dots}_{\varphi \vee \tau}.$$

Observing that  $s \neq 2 \wedge s' = s$  contradicts  $s' = 2$ , it follows that the left-hand side of the implication is false.

- Establishing  $\square \neg \psi$

We thus conclude the validity of

$$\psi_1 \Rightarrow \Diamond F.$$

In a symmetric way, we establish

$$\psi_2 \Rightarrow \Diamond F.$$

By the case splitting rule SPL-C, it follows that

$$\psi_1 \vee \psi_2 \Rightarrow \Diamond F.$$

Using the transitivity rule TRN-C with the previously established statement,  $\psi \Rightarrow \Diamond(\psi_1 \vee \psi_2)$ , we conclude

$$\psi \Rightarrow \Diamond F.$$

By the contradiction rule CONT, this establishes

$$\square \neg \psi. \blacksquare$$

An interesting point is that backward analysis is a mirror image of forward analysis, where the basic formula is the response formula  $p \Rightarrow \Diamond q$ . Forward analysis is a well accepted methodology for establishing response properties and is presented in the PROGRESS book. It bears a very close resemblance to backward analysis, except that instead of exploring possible predecessors we explore possible successors. The symmetry between backward exploration for safety and forward exploration for response is aesthetically pleasing and reduces the number of unrelated proof principles one has to consider.

## 4.6 Order-Preservation Properties

In this section we consider a more intricate safety property, one that is naturally expressed by causality formulas. This is the property of *order preservation*. In general, such a property relates the ordering of “output events” to the ordering of corresponding “input events,” and claims that ordering is preserved.

### Example (producer-consumer)

Reconsider program PROD-CONS of Fig. 4.3 (page 331). In Section 1.5 (page 133), we proved the invariance of

$$|send| \leq N$$

$$|ack| + |send| + at\_l_3 + at\_m_2 = N.$$

In Section 4.2 (page 331), we introduced the abbreviations

$$prod(u): at\_l_2 \wedge x = u$$

$$cons(u): at\_m_3 \wedge y = u$$

and established the no-invented-messages property, which is expressed by the causality formula

$$\psi_1: cons(u) \Rightarrow \Diamond prod(u),$$

and the intermediate formula used in its proof

$$\psi'_1: u \in send^* \Rightarrow \Diamond prod(u).$$

The *order-preservation* property for program PROD-CONS states that if  $u_1$  is consumed before  $u_2$ , then  $u_1$  must have been produced before  $u_2$ .

With the above abbreviations and under the no-duplicate-messages assumption, this property can be expressed by the formula:

$$\psi_2: \underbrace{cons(u_2) \wedge \Diamond cons(u_1)}_p \Rightarrow \Diamond \left( \underbrace{prod(u_2) \wedge \Diamond prod(u_1)}_r \right).$$

This formula states that

if at some instant  $j_2$  the value  $u_2$  is *consumed*, and the value  $u_1$  has been consumed at a preceding instant  $j_1$ ,  $j_1 \leq j_2$ , then there must also have been a previous instant  $i_2$ ,  $i_2 \leq j_2$ , at which  $u_2$  is *produced* and a still earlier instant  $i_1$ ,  $i_1 \leq i_2$ , in which  $u_1$  was produced.

Note that no ordering is implied between  $i_2$  and  $j_1$ , nor between  $i_1$  and  $j_1$ . In the absence of message duplication, the formula also states that messages are consumed in the same order in which they are produced.

Formula  $\psi_2$  has the form of a causality formula,

$$p \Rightarrow \Diamond r,$$

where  $p$  and  $r$  are past formulas. Note that this is the first example we consider where  $p$  and  $r$  are not state formulas. However, rule CAUS has been formulated for this more general case.

Wishing to prove the validity of  $\psi_2$  by rule CAUS, it only remains to construct the intermediate formula  $\varphi$ .

- *Generating  $\varphi$*

The formula  $\varphi$  should generalize  $p$  (i.e., be implied by  $p$ ), and should characterize all the situations immediately after  $r$ , up to the point that  $p$  holds. All these situations must bear evidence that  $u_1$  has been produced before  $u_2$  and therefore is consumed before  $u_2$ .

A first generalization of  $p$  is a situation in which  $u_1$  has already been consumed but  $u_2$  is still in the pipeline between production and consumption, i.e., in  $send^*$ , where the virtual variable  $send^*$  was defined in Section 4.2 as

$$send^* = [ \text{if } at\_m_{2,3} \text{ then } (y) \text{ else } \Lambda ] * send * [ \text{if } at\_l_3 \text{ then } (x) \text{ else } \Lambda ].$$

Such situations are covered by the formula

$$u_2 \in send^* \wedge \Diamond cons(u_1).$$

An even earlier situation, which still guarantees a consumption of  $u_1$  preceding a consumption of  $u_2$ , is when both  $u_1$  and  $u_2$  are in the pipeline, but  $u_1$  is closer to the front than  $u_2$ , unless they are equal. These situations are covered by the formula  $precede(u_1, u_2, send^*)$ , defined by

$$precede(u_1, u_2, send^*): \exists i, j: i \leq j: (send^*[i] = u_1 \wedge send^*[j] = u_2).$$

Note the *existential* character of this formula. It states that there exist an occurrence of  $u_1$  and an occurrence of  $u_2$ , the first preceding the second. This is consistent with the existential character of  $p$  (or  $r$ ), which states the existence of an occurrence of consumption (or production) of  $u_2$  which is preceded by some consumption (or production) of  $u_1$ . Note that  $precede(u_1, u_2, send^*)$  implies that  $u_1, u_2 \in send^*$ .

We thus propose to prove the order-preservation property expressed by the causality formula

$$\psi_2: cons(u_2) \wedge \Diamond cons(u_1) \Rightarrow \Diamond (prod(u_2) \wedge \Diamond prod(u_1))$$

using the intermediate formula

$$\varphi: (u_2 \in send^* \wedge \Diamond cons(u_1)) \vee precede(u_1, u_2, send^*).$$

Let us show that this proposed formula satisfies the three premises of rule CAUS.

- *Checking C1*

Premise C1 states:

$$\underbrace{cons(u_2) \wedge \Diamond cons(u_1)}_p \Rightarrow \underbrace{(u_2 \in send^* \wedge \Diamond cons(u_1)) \vee \dots}_\varphi \vee \underbrace{\dots}_r.$$

It is valid since, by the definition of  $send^*$ , it follows that

$$cons(u_2) \Rightarrow u_2 \in send^*.$$

In fact,  $cons(u_2)$  implies that  $u_2 = hd(send^*)$ .

- *Checking C2*

Premise C2 states:

$$\underbrace{\dots \wedge send^* = \Lambda \wedge \dots}_\Theta \rightarrow$$

$$\neg \left( \underbrace{(u_2 \in send^* \wedge \Diamond cons(u_1)) \vee precede(u_1, u_2, send^*)}_\varphi \right)_0 \vee \underbrace{\dots}_{(\tau)_0}.$$

It is valid since  $\Theta$  implies  $send^* = \Lambda$ , from which  $u_1 \notin send^*$  and  $u_2 \notin send^*$  follow. Consequently,  $(\varphi)_0$  is false, and its negation is true.

- *Checking C3*

For each  $\tau$ , we have to show

$$\rho_\tau \wedge \left( \underbrace{(u_2 \in send^* \wedge \Diamond cons(u_1)) \vee precede(u_1, u_2, send^*)}_\varphi \right)' \Rightarrow$$

$$\underbrace{(u_2 \in send^* \wedge \Diamond cons(u_1)) \vee precede(u_1, u_2, send^*)}_\varphi \vee$$

$$\underbrace{prod(u_2) \wedge \Diamond prod(u_1)}_r.$$

We have to consider only two critical transitions:  $\ell_2$  and  $m_3$ , the only transitions that can modify  $\varphi$ . The reader is requested to complete this proof in **Problem 4.8**.

This establishes the validity of the causality formula  $\psi_2$ . ■

## The Case of Duplicate Messages

In page 333, we discussed the no-invented-messages property in the case of duplicate messages, and realized that it is necessary to require the stronger property of no-invented-replicates. Here again, we face a similar difficulty in specifying order preservation if duplicate messages are allowed. The problem is that not every sequence of produce-consume events satisfying the formula

$$\psi_2: \text{cons}(u_2) \wedge \Diamond \text{cons}(u_1) \Rightarrow \Diamond(\text{prod}(u_2) \wedge \Diamond \text{prod}(u_1))$$

meets our expectations of an order-preserving sequence.

Consider, for example, the following sequence:

$$\text{prod}(1), \quad \text{prod}(2), \quad \text{prod}(1), \quad \text{cons}(1), \quad \text{cons}(1), \quad \text{cons}(2).$$

The sequence obviously does not have the order-preservation property, since the second message 1 was produced after message 2, while it was consumed before message 2. Yet, this sequence satisfies formula  $\psi_2$ . Obviously, we also need a stronger property here, to which we refer as the *order-replication* property.

As for the no-invented-replicates property, this difficulty can be overcome by one of the three considered solutions:

- Disallowing duplicate messages (the approach taken in this chapter).
- Considering data-independent programs.
- Using auxiliary variables or quantified formulas.

In **Problem 4.9**, we ask the reader to formulate and prove a claim similar to Claim 4.3 which states that formula  $\psi_2$  specifies order replication for data-independent programs. In **Problem 4.10**, we ask the reader to write a quantified temporal formula which specifies order replication without assuming data independence.

## 4.7 History Variables

The proofs of many properties, in particular those involving several temporal reference points such as the previously considered order-preservation property, can be greatly simplified by the introduction of special auxiliary variables called *history variables*. The idea underlying history variables is the important notion of *statifying* the past, i.e., introducing auxiliary variables whose values in the current state reflect all the facts we need to know about the past.

### Example (producer-consumer)

We will illustrate the use of history variables on the producer-consumer program PROD-CONS-H presented in Fig. 4.11. This version is obtained by augmenting program PROD-CONS of Fig. 4.3 (page 331) with two auxiliary variables,  $h_p$  and  $h_c$ .

Both variables assume values that are lists of integers (assuming the produced and consumed values are integers). The variable  $h_p$  is the history variable recording the sequence of *produced* values in their order of production. The variable  $h_c$  is the history variable recording the sequence of *consumed* values in their order

```

local      send, ack: channel [1..] of integer
          where send =  $\Lambda$ , ack =  $\underbrace{[1, 1, \dots, 1]}_N$ 
auxiliary  $h_p, h_c$  : list of integer
          where  $h_p = \Lambda, h_c = \Lambda$ 

Prod :: 
$$\left[ \begin{array}{l} \text{local } x, t: \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \langle \text{produce } x \rangle \\ h_p := h_p \bullet x \end{array} \right] \\ \ell_2: \text{ack} \Rightarrow t \\ \ell_3: \text{send} \Leftarrow x \end{array} \right]$$

||

Cons :: 
$$\left[ \begin{array}{l} \text{local } y: \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{send} \Rightarrow y \\ m_2: \langle \text{ack} \Leftarrow 1 \\ h_c := h_c \bullet y \end{array} \right] \\ m_3: \text{consume } y \end{array} \right]$$


```

Fig. 4.11. Program PROD-CONS-H (producer-consumer) with history variables.

of consumption. Inspecting program PROD-CONS-H, it is easy to see that, indeed, the variables  $h_p$  and  $h_c$  record the sequence of values produced and consumed.

One may wonder why appending  $x$  to the end of  $h_p$  is grouped together with the **produce** statement, while the operation of appending  $y$  to  $h_c$  is grouped with the statement preceding the **consume**. This is a result of the way we defined the abbreviations *prod* and *cons*. These abbreviations identify a producing state as one in which control of the producing process is at  $\ell_2$ , which is the location *following* the **produce** statement. This is necessary because, otherwise, we do not have yet the new value of  $x$ . On the other hand, the consuming location was identified as  $m_3$ , the location *before* the **consume** statement. To achieve complete symmetry between the two, we could have introduced an additional

location following the **consume** statement and identified it as the consuming location.

These facts can be used to relate past events to the present contents of the variables  $h_p$  and  $h_c$ . The relations are stated by the invariance of the formulas:

$$\chi_1: \diamondsuit \text{cons}(u) \leftrightarrow u \in h_c$$

$$\chi_2: \diamondsuit \text{prod}(u) \leftrightarrow u \in h_p.$$

These invariants state that at any state in the computation, there was a preceding event of consumption (or production) of the value  $u$  iff  $u$  is a member of  $h_c$  (or  $h_p$ ).

Similarly, consider the following invariants:

$$\chi_3: \diamondsuit(\text{cons}(u_2) \wedge \diamondsuit \text{cons}(u_1)) \leftrightarrow \text{precede}(u_1, u_2, h_c)$$

$$\chi_4: \diamondsuit(\text{prod}(u_2) \wedge \diamondsuit \text{prod}(u_1)) \leftrightarrow \text{precede}(u_1, u_2, h_p).$$

Invariant  $\chi_3$  states that the events of consumption of  $u_2$  and consumption of  $u_1$  have already happened, with a consumption of  $u_1$  preceding a consumption of  $u_2$ , iff both  $u_1$  and  $u_2$  appear in the list  $h_c$ , with an appearance of  $u_1$  preceding an appearance of  $u_2$ . Invariant  $\chi_4$  describes a similar connection between the occurrences of production events and appearances of the corresponding values in the list  $h_p$ .

We refer to these four invariants as *statification invariants*, since each of them identifies a state formula that holds at a position iff a corresponding past formula holds there.

- *Proving the causality and order-preservation properties*

It is now straightforward to express the safety properties in which we are interested as state formulas that refer to the history variables. We persist with the assumption of non-duplicate messages.

The no-invented-messages property stating that

every value consumed has been previously produced,

can be expressed by the invariance of the assertion

$$\widehat{\psi}_1: u \in h_c \rightarrow u \in h_p.$$

Since the variables  $h_c$  and  $h_p$  contain the full consumption and production histories, it is not necessary to use any temporal operators and it is sufficient to compare the two at any state.

The order-preservation property states that

if  $u_1$  has been consumed before  $u_2$ ,

then there is a  $u_1$  that was produced before some  $u_2$ .

This property can be expressed by the invariance of the assertion

$$\widehat{\psi}_2: \text{precede}(u_1, u_2, h_c) \rightarrow \text{precede}(u_1, u_2, h_p).$$

To prove these properties, we introduce the virtual variable  $\text{send}^\sharp$ , defined by:

$$\text{send}^\sharp = [\text{if } \text{at\_}m_2 \text{ then } (y) \text{ else } \Lambda] * \text{send} * [\text{if } \text{at\_}\ell_{2,3} \text{ then } (x) \text{ else } \Lambda].$$

The variable  $\text{send}^\sharp$  differs from the previously used virtual variable  $\text{send}^*$  in that, on being at  $m_3$ ,  $y$  is still a member (the first element) of  $\text{send}^*$ , but is no longer a member of  $\text{send}^\sharp$ . Another difference is that, being at  $\ell_2$ ,  $x \in \text{send}^\sharp$  but  $x \notin \text{send}^*$ .

It is easy to ascertain that the only two transitions that modify the value of  $\text{send}^\sharp$  are  $\ell_1$  and  $m_2$ . The effect of these transitions is given by:

$$\rho_{\ell_1} \rightarrow (\text{send}^\sharp)' = \text{send}^\sharp * x'$$

$$\rho_{m_2} \rightarrow (\text{send}^\sharp)' = \text{tl}(\text{send}^\sharp).$$

As we will show below, the two safety properties  $\widehat{\psi}_1$  and  $\widehat{\psi}_2$  can be deduced from the invariance of the single assertion:

$$\widehat{\psi}: h_p = h_c * \text{send}^\sharp.$$

This assertion states that the list  $h_p$  is always equal to the concatenation of the lists  $h_c$  and  $\text{send}^\sharp$ .

- $\widehat{\psi}$  is invariant

It is relatively easy to verify by rule INV-B that  $\widehat{\psi}$  is an inductive assertion. Since  $\Theta$  implies  $h_p = h_c = \text{send}^\sharp = \Lambda$ , it follows that  $\widehat{\psi}$  holds initially.

Consider the two transitions that may affect  $h_p$ ,  $h_c$ , or  $\text{send}^\sharp$ .

- Transition  $\ell_1$

The verification condition for  $\ell_1$  is

$$\dots \wedge h'_p = h_p * x' \wedge h'_c = h_c \wedge (\text{send}^\sharp)' = \text{send}^\sharp * x' \wedge \dots \wedge \underbrace{\rho_{\ell_1}}_{h_p = h_c * \text{send}^\sharp}.$$

$$\underbrace{h_p = h_c * \text{send}^\sharp}_{\widehat{\psi}} \rightarrow \underbrace{h'_p = h'_c * (\text{send}^\sharp)' * x'}_{\widehat{\psi}'}$$

It follows that, under  $\rho_{\ell_1}$ , it is sufficient to prove

$$h_p = h_c * \text{send}^\sharp \rightarrow h_p * x' = h_c * (\text{send}^\sharp * x').$$

Using the properties of list concatenation and element appending, it is enough to show

$$h_p = h_c * \text{send}^\sharp \rightarrow h_p * x' = (h_c * \text{send}^\sharp) * x',$$

which is obviously valid.

- Transition  $m_2$

The verification condition for  $m_2$  is

$$\overbrace{\dots \wedge at\_m_2 \wedge h'_c = h_c * y \wedge h'_p = h_p \wedge (send^\#)' = tl(send^\#) \wedge \dots}^{\rho_{m_2}} \wedge$$

$$\underbrace{h_p = h_c * send^\#}_{\widehat{\psi}} \rightarrow \underbrace{h'_p = h'_c * (send^\#)'}_{\widehat{\psi}'}$$

By  $at\_m_2$  and the definition of  $send^\#$ , we have  $y = hd(send^\#)$ . It follows that  $h'_c * (send^\#)' = (h_c * hd(send^\#)) * tl(send^\#) = h_c * send^\# = h_p = h'_p$ .

This means that  $m_2$  removes the first element from  $send^\#$ , and appends it to the end of  $h_c$ . This certainly preserves the concatenation  $h_c * send^\#$ .

- $\widehat{\psi}_1$  and  $\widehat{\psi}_2$  are invariants

Once we have established the invariance of

$$\widehat{\psi}: h_p = h_c * send^\#,$$

it is straightforward to establish that  $\widehat{\psi}_1$  and  $\widehat{\psi}_2$  are invariants.

The invariance of  $\widehat{\psi}$  implies the prefix-inclusion property, stating that

$$h_c \sqsubseteq h_p$$

holds at all positions of the computation. That is, the sequence of consumed values is always a prefix of the sequence of produced values. Prefix inclusion can also be expressed in terms of the elements of the two lists, and it states that

$$|h_c| \leq |h_p| \quad \text{and} \quad \forall i: i \leq |h_c|: h_p[i] = h_c[i].$$

We show that prefix inclusion implies the two properties we wish to prove.

The causality property

$$\widehat{\psi}_1: u \in h_c \rightarrow u \in h_p,$$

is obvious, since by prefix inclusion, if  $h_c[i] = u$ , for some  $i \leq |h_c|$ , then clearly  $i \leq |h_p|$  and  $h_p[i] = u$ .

Similarly, it is easy to show the order-preservation property

$$\widehat{\psi}_2: precede(u_1, u_2, h_c) \rightarrow precede(u_1, u_2, h_p).$$

For  $precede(u_1, u_2, h_c)$  to hold, there must be indices  $i$  and  $j$ ,  $i \leq j \leq |h_c|$ , such that  $h_c[i] = u_1$ ,  $h_c[j] = u_2$ . By prefix inclusion,  $i \leq j \leq |h_p|$ ,  $h_p[i] = u_1$ , and  $h_p[j] = u_2$ , establishing  $precede(u_1, u_2, h_p)$ .

- *Proving the statification relations*

The only missing link for completion of the proof is showing the equivalence of the past formulas to the state formulas referring to the history variables, as stated by  $\chi_1 \cdots \chi_4$ .

In Problem 4.11, the reader is requested to prove the invariance of these formulas over program PROD-CONS-H.

In many cases, the relation between the past formulas and the history variables is so obvious that we forego its formal proof. Thus, in the example above, we would normally have stated the obvious statification equivalences,  $\chi_1 \cdots \chi_4$ , without proof, and then proceeded to show the invariance of

$$\widehat{\psi}: h_p = h_c * \text{send}^\#.$$

From this invariance we would then as above derive the invariant properties

$$\widehat{\psi}_1: u \in h_c \rightarrow u \in h_p$$

$$\widehat{\psi}_2: \text{precede}(u_1, u_2, h_c) \rightarrow \text{precede}(u_1, u_2, h_p).$$

■

## **Allowing Message Duplication**

In the previous verification of program PROD-CONS-H, we used the invariant

$$h_c \sqsubseteq h_p$$

to prove the invariance of  $\widehat{\psi}_1$  and  $\widehat{\psi}_2$  over program PROD-CONS-H.

Assertions  $\widehat{\psi}_1$  and  $\widehat{\psi}_2$  specify the properties of no-invented-messages and order preservation, but only under the assumption of non-duplicate messages. However, the assertion  $\widehat{\psi}: h_p = h_c * \text{send}^\#$  and its consequence  $h_c \sqsubseteq h_p$  are invariant also when messages are duplicated. The important point is that the invariant  $h_c \sqsubseteq h_p$  ensures the stronger properties of no-invented-replicates and order replication in their full generality, taking duplicate messages into account.

Assertion  $h_c \sqsubseteq h_p$  implies the no-invented-replicates property in the form

the  $n$ th consumption of a message is preceded by at least  $n$  productions of the same message.

To see this, consider the situation just after the  $n$ th consumption of message  $u$ . This means that  $u$  is the last element in the list  $h_c$  and, altogether,  $h_c$  contains  $n$  copies of  $u$ . Since  $h_c \sqsubseteq h_p$  is an invariant, this implies that  $h_p$  contains at least  $n$  copies of  $u$ . Since  $h_p$  summarizes the history of productions up to the present, it follows that  $u$  has been produced at least  $n$  times in the past.

Assertion  $h_c \sqsubseteq h_p$  also implies the property of order replication in the form

if the  $n_1$ th consumption of  $u_1$  precedes the  $n_2$ th consumption of  $u_2$ ,

then the  $n_1$ th production of  $u_1$  must precede the  $n_2$ th production of  $u_2$ .

To see this, consider the situation just after the  $n_2$ th consumption of  $u_2$ . In this case,  $h_c$  has the form

$$h_c: \underbrace{\dots}_{n_1 \text{ copies of } u_1} \overbrace{u_1}^{} \dots \underbrace{\dots}_{n_2 \text{ copies of } u_2} u_2 .$$

By  $h_c \sqsubseteq h_p$ ,  $h_c$  is a prefix of  $h_p$ . We conclude that the  $n_1$ th production of  $u_1$  precedes the  $n_2$ th production of  $u_2$ .

## 4.8 Back-to Rule

In Section 3.2 we illustrated the specification and analysis of overtaking properties by (*nested*) *waiting-for* formulas of the form

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \dots q_1 \mathcal{W} q_0,$$

for assertions  $p, q_0, \dots, q_m$ . We proposed rule NWAIT of Fig. 3.6 for establishing such waiting-for formulas. In the same way that rule INV-P extends rule INV, it is straightforward to extend rule NWAIT to handle the case where  $p, q_0, \dots, q_m$  are past formulas. All that is required is to replace all implications by entailments.

In this section, we consider *nested back-to* formulas of the form

$$p \Rightarrow q_m \mathcal{B} q_{m-1} \dots q_1 \mathcal{B} q_0,$$

for past formulas  $p, q_0, \dots, q_m$ . To establish properties expressible by such formulas, we will present a rule NBACK.

### Back-to vs. Waiting-for

The basic symmetry between nested waiting-for formulas and nested back-to formulas is stated by the equivalence

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \dots q_1 \mathcal{W} r \quad \sim \quad (\neg q_1) \Rightarrow q_2 \mathcal{B} q_3 \dots q_m \mathcal{B} (\neg p) \mathcal{B} r.$$

The nested waiting-for formula  $p \Rightarrow q_m \mathcal{W} q_{m-1} \dots q_1 \mathcal{W} r$  states that every  $p$  is followed by a succession of a  $q_m$ -interval, a  $q_{m-1}$ -interval, and so on, which can be terminated only by an occurrence of  $r$ . The equivalence shows that this property is a safety property that also states that every occurrence of  $\neg q_1$  is preceded by a succession of a  $q_2$ -interval, preceded by a  $q_3$ -interval, and so on, until a  $\neg p$  interval. The sequence may be interrupted earlier or terminated by an occurrence of  $r$ .

For the special case  $n = 1$ , the equivalence reduces to

$$p \Rightarrow q \mathcal{W} r \quad \sim \quad (\neg q) \Rightarrow (\neg p) \mathcal{B} r.$$

Taking  $r = F$ , we obtain

$$p \Rightarrow \square q \quad \sim \quad (\neg q) \Rightarrow \Box(\neg p).$$

With these equivalences, it is easy to express any property, specified by a waiting-for formula, by a back-to formula.

Every nested back-to formula is of the form  $\square \psi$  where  $\psi$  is a past formula and therefore is a canonical safety formula. The equivalence between nested waiting-for and nested back-to formulas shows that all nested waiting-for formulas are safety formulas too (although not in canonical form).

In **Problem 4.12**, the reader is requested to prove the equivalence between the nested waiting-for and the nested back-to formulas.

### Example (Peterson's algorithm for mutual exclusion — version 1)

Consider the following precedence property for program MUX-PET1 of Fig. 4.12 (see Fig. 3.4, page 255),

$$\underbrace{at_{\ell_3} \wedge at_{m_{0..2}}}_{p} \Rightarrow \underbrace{\neg at_{m_4}}_q \mathcal{W} \underbrace{at_{\ell_4}}_r.$$

This formula states that if  $P_1$  precedes  $P_2$  in getting to the await statement  $\ell_3$ , then it also precedes  $P_2$  in entering the critical section, represented by the statements  $\ell_4, m_4$ , respectively.

Using the equivalence presented above, this waiting-for formula is equivalent to the back-to formula

$$\psi_1: \underbrace{at_{m_4}}_{\neg q} \Rightarrow \underbrace{\neg(at_{\ell_3} \wedge at_{m_{0..2}})}_{\neg p} \mathcal{B} \underbrace{at_{\ell_4}}_r.$$

This formula states that if we observe  $P_2$  to be at the critical section (i.e.,  $P_2$  preceded  $P_1$  in entering the critical section), then since  $P_1$ 's last visit to its own critical section at  $\ell_4$ ,  $P_1$  has never been in a state where it had priority over  $P_2$  (i.e.,  $at_{\ell_3} \wedge at_{m_{0..2}}$ ).

While this is the direct equivalent of the waiting-for formula, it is not necessarily the most natural past formula that expresses the desired precedence property. A somewhat more natural expression of the property is given by the back-to formula

$$\psi_2: at_{m_4} \Rightarrow at_{m_{3,4}} \mathcal{B} (at_{m_3} \wedge at_{\ell_{0..2}}).$$

This formula states that  $P_2$  preceding  $P_1$  in entering the critical section, can happen only if  $P_2$  has been in the critical section or at the waiting position

**local**  $y_1, y_2$ : boolean where  $y_1 = F, y_2 = F$   
 $s$  : integer where  $s = 1$

$P_1 :: \left[ \begin{array}{l} \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: (y_1, s) := (\top, 1) \\ \ell_3: \text{await } \neg y_2 \vee s \neq 1 \\ \ell_4: \text{critical} \\ \ell_5: y_1 := F \end{array} \right] \end{array} \right]$

$P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: (y_2, s) := (\top, 2) \\ m_3: \text{await } \neg y_1 \vee s \neq 2 \\ m_4: \text{critical} \\ m_5: y_2 := F \end{array} \right] \end{array} \right]$

Fig. 4.12. Program MUX-PET1 (Peterson's algorithm) — version 1.

continuously, ever since a point in which it had priority over  $P_1$ , i.e.,  $\text{at\_}m_3 \wedge \text{at\_}\ell_{0..2}$ .

The two formulas  $\psi_1$  and  $\psi_2$  are not logically equivalent, but in the context of program MUX-PET1 they state similar properties. ■

## Back-to Formulas

Rule BACK presented in Fig. 4.13 can be used to establish properties expressed by back-to formulas. Premise B1 states that any position satisfying  $p$  must satisfy either the goal  $r$  or the intermediate formula  $\varphi$ . Premise B3 states that any position satisfying  $\varphi$  must be preceded by a position satisfying  $\varphi \vee r$ . Premise B2 states that the intermediate formula  $\varphi$  entails  $q$ .

We show that the rule can be obtained by symmetry from rule WAIT-P, presented in Fig. 4.14. This rule is a version of rule WAIT of Fig. 3.3 (page 254), generalized to the case that the formulas appearing in the rule are past formulas.

For past formulas  $p, q, r, \varphi$ ,

$$\begin{array}{l} \text{B1. } p \Rightarrow \varphi \vee r \\ \text{B2. } \varphi \Rightarrow q \\ \text{B3. } \{\varphi \vee r\} T^{-1} \{\varphi\} \\ \hline p \Rightarrow q \mathcal{B} r \end{array}$$

Fig. 4.13. Rule BACK (back-to).

For past formulas  $u, v, w, \chi$ ,

$$\begin{array}{l} \text{U1. } u \Rightarrow \chi \vee w \\ \text{U2. } \chi \Rightarrow v \\ \text{U3. } \{\chi\} T \{\chi \vee w\} \\ \hline u \Rightarrow v \mathcal{W} w \end{array}$$

Fig. 4.14. Rule WAIT-P (past waiting-for).

**Justification** The soundness of rule BACK can be established directly on semantic grounds, in a way similar to the justification of the other rules. However, it is interesting to show that rule BACK can be derived from rule WAIT-P.

Assume that we have formulas satisfying premises B1–B3. We will use rule WAIT-P to prove

$$\neg q \Rightarrow (\neg p) \mathcal{W} r,$$

which by the claim above is equivalent to

$$p \Rightarrow q \mathcal{B} r.$$

Taking formulas  $u, v$ , and  $w$ , appearing in the rule to be  $\neg q$ ,  $\neg p$ , and  $r$ , respectively, we have to find a past formula  $\chi$ , such that the following three premises are satisfied:

- U1.  $\neg q \Rightarrow \chi \vee r$
- U2.  $\chi \Rightarrow \neg p$
- U3.  $\{\chi\} T \{\chi \vee r\}.$

We choose

$$\chi: \neg\varphi \wedge \neg r,$$

and proceed to show how we can derive premises U1–U3 from the given B1–B3.

- *Establishing U1*

By B2, we have  $\varphi \Rightarrow q$ . By contraposition, this is equivalent to

$$\neg q \Rightarrow \neg\varphi.$$

This entailment can be weakened to

$$\neg q \Rightarrow \neg\varphi \vee r,$$

which is equivalent to

$$\neg q \Rightarrow \underbrace{\neg\varphi \wedge \neg r}_{\chi} \vee r.$$

This establishes U1.

- *Establishing U2*

By B1, we have  $p \Rightarrow \varphi \vee r$ , which by contraposition yields

$$\underbrace{\neg\varphi \wedge \neg r}_{\chi} \Rightarrow \neg p.$$

This establishes U2.

- *Establishing U3*

By B3, we have for every  $\tau \in T$ ,

$$\rho_\tau \wedge \varphi' \Rightarrow \varphi \vee r.$$

This is equivalent to

$$\rho_\tau \wedge (\neg\varphi \wedge \neg r) \Rightarrow \neg\varphi'.$$

This entailment can be weakened to

$$\rho_\tau \wedge (\neg\varphi \wedge \neg r) \Rightarrow \neg\varphi' \vee r',$$

which is equivalent to

$$\rho_\tau \wedge \underbrace{\neg\varphi \wedge \neg r}_{\chi} \Rightarrow \underbrace{\neg\varphi' \wedge \neg r'}_{\chi'} \vee r'.$$

This establishes U3.

Thus, by rule WAIT-P, we may deduce

$$\neg q \Rightarrow (\neg p) \mathcal{W} r. \blacksquare$$

We use rule BACK in the following example.

**Example** (Peterson's algorithm — version 1)

Consider again program MUX-PET1 (Fig. 4.12, page 365) and let us use rule BACK to prove the precedence property

$$\psi_2: \underbrace{at\_m_4}_p \Rightarrow \underbrace{at\_m_{3,4}}_q \stackrel{\mathcal{B}}{\rightarrow} \underbrace{at\_m_3 \wedge at\_\ell_{0..2}}_r.$$

To apply rule BACK, we need to find an assertion  $\varphi$  that satisfies the three premises B1–B3. This assertion should capture the situations at which  $P_2$  is either at  $m_4$  or at  $m_3$  with priority over  $P_1$ . When  $P_2$  is at  $m_3$ , it has priority over  $P_1$ , if either  $P_1$  is in the range  $\ell_{0..2}$  or  $P_1$  is at  $\ell_3$  with  $s = 1$ . Since the first case already attains the goal  $at\_m_3 \wedge at\_\ell_{0..2}$ , the natural candidate for the intermediate situation is:

$$\varphi: at\_m_4 \vee (at\_m_3 \wedge s = 1).$$

We will show that the three premises of the rule hold for this choice of  $\varphi$ .

Premise B1 reads

$$\underbrace{at\_m_4}_p \Rightarrow \underbrace{at\_m_4 \vee \dots}_{\varphi} \vee \underbrace{\dots}_r,$$

which is obvious.

Premise B2 reads

$$\underbrace{at\_m_4 \vee (at\_m_3 \wedge s = 1)}_{\varphi} \Rightarrow \underbrace{at\_m_{3,4}}_q,$$

which is also obvious.

Premise B3 requires, for all transitions  $r \in \mathcal{T}$

$$\rho_r \wedge \underbrace{at'_m_4 \vee (at'_m_3 \wedge s' = 1)}_{\varphi'} \Rightarrow \underbrace{at\_m_4 \vee (at\_m_3 \wedge s = 1)}_{\varphi} \vee \underbrace{at\_m_3 \wedge at\_\ell_{0..2}}_r.$$

Let us consider transitions which may potentially change the value of  $\varphi$  from F to T, and thereby violate the entailment above. The candidate transitions are  $\ell_2$  while  $at\_m_3$  holds,  $m_2$ , and  $m_3$ .

- Transition  $\ell_2$  while  $at\_m_3$  holds

Transition  $\ell_2$  is enabled in an  $at\_m_3$ -state only when  $at\_m_3 \wedge at\_\ell_{0..2}$  holds, establishing  $r$ .

- Transition  $m_2$

This transition sets  $at'_m_3 = T$  and  $s' = 2$ , so  $\varphi'$  cannot hold.

- Transition  $m_3$

This transition is enabled only from a state in which  $at\_m_3 \wedge (\neg y_1 \vee s = 1)$  holds.

If  $y_1 = F$ , then by the invariant  $y_1 \leftrightarrow at\_\ell_{3..5}$  (established for MUX-PETI in Section 1.4),  $at\_\ell_{0..2}$  must hold, establishing  $r$ .

If  $y_1 = T$ , then  $s = 1$ , and therefore  $at\_m_3 \wedge s = 1$  holds, establishing  $\varphi$ .

This establishes the validity of

$$\psi_2: at\_m_4 \Rightarrow at\_m_{3,4} \mathcal{B} (at\_m_3 \wedge at\_\ell_{0..2}). \blacksquare$$

### Nested Back-to Formulas

It is easy to derive a rule for nested back-to formulas, rule NBACK (Fig. 4.15). The rule can be derived by symmetry from rule NWAIT (Fig. 3.6), for nested waiting-for formulas, with the appropriate generalization to past formulas instead of assertions.

For past formulas  $p, q_0, q_1, \dots, q_m$ , and  $\varphi_0, \varphi_1, \dots, \varphi_m$ ,

$$\begin{array}{ll}
 \text{N1. } p \Rightarrow \bigvee_{j=0}^m \varphi_j & \\
 \text{N2. } \varphi_i \Rightarrow q_i & \text{for } i = 0, 1, \dots, m \\
 \text{N3. } \{\varphi_i\} T^{-1} \left\{ \bigvee_{j \leq i} \varphi_j \right\} & \text{for } i = 1, \dots, m \\
 \hline
 p \Rightarrow q_m \mathcal{B} q_{m-1} \cdots q_1 \mathcal{B} q_0 &
 \end{array}$$

Fig. 4.15. Rule NBACK (nested back-to).

Note that premise N3, when expanded, reads

$$\rho_T \wedge \varphi'_i \Rightarrow \bigvee_{j \leq i} \varphi_j$$

for every  $\tau \in T$  and  $i = 1, \dots, m$ .

In Problem 4.13, the reader is requested to show how rule NBACK can be derived from rule NWAIT, and the equivalences listed in this section.

**Example** (Peterson's algorithm — version 2)

Let us prove the property of 1-bounded overtaking from  $\ell_4$  for program MUX-PET2 (Fig. 4.10).

This property can be expressed by the following nested back-to formula:

$$\underbrace{at\_m_5}_p \Rightarrow \underbrace{at\_m_5}_{q_2} \mathcal{B} \underbrace{\neg at\_m_5}_{q_1} \mathcal{B} \underbrace{\neg at\_\ell_4}_{q_0}.$$

The formula states that if we are currently at  $m_5$ , then we have been at  $m_5$  for a while, preceded by an interval of  $\neg at\_m_5$ . The  $\neg at\_m_5$  interval cannot be preceded by an ( $at\_m_5$ )-state without encountering a state at which  $P_1$  is no longer (or not yet) waiting at  $\ell_4$ . Thus, the formula implies that  $P_2$  cannot visit  $m_5$  twice in succession, while  $P_1$  is continuously waiting at  $\ell_4$ .

To prove this property by rule NBACK, we have to find assertions  $\varphi_0, \varphi_1, \varphi_2$ , such that:

$$N1. \quad at\_m_5 \rightarrow \varphi_2 \vee \varphi_1 \vee \varphi_0$$

$$N2. \quad \varphi_0 \rightarrow \neg at\_\ell_4$$

$$\varphi_1 \rightarrow \neg at\_m_5$$

$$\varphi_2 \rightarrow at\_m_5$$

$$N3 \ (i=2) \ \{\varphi_2\} T^{-1} \{\varphi_2 \vee \varphi_1 \vee \varphi_0\}$$

$$(i=1) \ \{\varphi_1\} T^{-1} \{\varphi_1 \vee \varphi_0\}.$$

In particular, the first formula for premise N3 (case  $i = 2$ ) identifies the predecessors of  $\varphi_2$ -states as states satisfying  $\varphi_0, \varphi_1$ , or  $\varphi_2$ .

A natural choice is to take

$$\varphi_2: at\_m_5 \quad \text{and} \quad \varphi_0: \neg at\_\ell_4.$$

The only states which do not satisfy  $\varphi_2$ , but which are predecessors of  $\varphi_2$ -states, are those that satisfy

$$at\_m_4 \wedge (\neg y_1 \vee s = 1).$$

Among those, the states which satisfy  $\neg y_1$  must, by the previously established invariant  $y_1 \leftrightarrow at\_\ell_{3..6}$  (page 124), satisfy  $\neg at\_\ell_4$ , that is,  $\varphi_0$ . It is therefore reasonable to take  $\varphi_1$  as

$$\varphi_1: at\_m_4 \wedge s = 1.$$

Let us show that each of the premises of rule NBACK is satisfied by this choice.

- Premises N1 and N2

Premise N1

$$at\_m_5 \rightarrow \underbrace{at\_m_5}_{\varphi_2} \vee \dots \vee \dots$$

is trivially satisfied.

The two implications of N2 involving  $\varphi_0$  and  $\varphi_2$  are trivial. The implication for  $\varphi_1$  reads

$$\underbrace{at\_m_4 \wedge s = 1}_{\varphi_1} \rightarrow \neg at\_m_5,$$

and is obviously valid.

Let us consider premise N3 for  $i = 2$  and  $i = 1$ .

- Premise N3 ( $i = 2$ )

The inverse verification condition requires

$$\rho_T \wedge \underbrace{at'_m_5}_{\varphi'_2} \rightarrow \underbrace{at\_m_5}_{\varphi_2} \vee \underbrace{at\_m_4 \wedge s = 1}_{\varphi_1} \vee \underbrace{\neg at\_l_4}_{\varphi_0}.$$

Clearly, the only relevant transition that may change the value of  $at\_m_5$  from F to T is  $m_4$ . Obviously,  $\rho_{m_4}$  implies  $at\_m_4$  and  $\neg y_1 \vee s = 1$ . Since  $\neg y_1 \rightarrow \neg at\_l_4$  by the known invariant  $y_1 \leftrightarrow at\_l_{3..6}$ , it follows that  $\rho_{m_4}$  implies the disjunction  $(at\_m_4 \wedge s = 1) \vee \neg at\_l_4$ , establishing the validity of the verification condition.

- Premise N3 ( $i = 1$ )

The inverse verification condition requires:

$$\rho_T \wedge \underbrace{at'_m_4 \wedge s' = 1}_{\varphi'_1} \rightarrow \dots \vee \underbrace{\neg at\_l_4}_{\varphi_0}.$$

There are only two transitions that can possibly change the value of  $at\_m_4 \wedge s = 1$  from F to T. They are  $m_3$  and  $l_3$ .

In the case of  $m_3$ , we have that  $\rho_{m_3}$  implies  $s' = 2$ , which makes  $\rho_{m_3} \wedge \varphi'_1$  false, and the implication valid.

In the case of  $l_3$ , we have that  $\rho_{l_3}$  implies  $at\_l_3$ , which implies  $\varphi_0: \neg at\_l_4$ .

This establishes a bound of one on the possible overtaking from  $l_4$ , stated by a nested back-to formula. ■

In **Problem 4.14**, the reader is requested to prove a bound of two on the possible overtaking from  $l_3$  for the same program MUX-PET2 (Fig. 4.10). This property can be stated by the following nested back-to formula:

$$at\_m_5 \Rightarrow at\_m_5 \mathcal{B} (\neg at\_m_5) \mathcal{B} at\_m_5 \mathcal{B} (\neg at\_m_5) \mathcal{B} \neg at\_l_{3..4}.$$

In **Problem 4.15**, the reader is requested to prove the nested back-to formula

$$y = u \Rightarrow (y = u) \mathcal{B} (u \in \text{send}) \mathcal{B} (x = u) \mathcal{B} \text{ prod}(u)$$

for program PROD-CONS of Fig. 4.3. This formula traces backwards the history of the value  $u$  from consumption to production.

## \* 4.9 Completeness

In this section we show that rule INV-P is complete for proving the validity of formulas of the form  $\Box \psi$  for arbitrary past formula  $\psi$ . This shows that rule INV-P is adequate for proving  $P$ -validity of all safety formulas. As in the preceding two completeness theorems for rules INV (Section 2.5) and NWAIT (Section 3.5), completeness is relative to first-order reasoning. The precise statement of completeness is given by the following theorem.

**Theorem 4.5** (completeness of rule INV-P)

For every program  $P$  and past formula  $\psi$  such that  $\Box \psi$  is  $P$ -valid, there exists a past formula  $\varphi$  such that the premises of rule INV-P are provable from state (first-order) validities.

Since premises P1 and P3 of rule INV-P are entailments, it is no longer sufficient to claim that they are valid since this does not yet yield full reduction to first-order validities. Instead, the completeness theorem states that all premises are provable (using rules such as instantiation, instantiated generalization, temporalization, and entailment reasoning) from state validities.

The proof of completeness is based on *reduction*. First, we show how to reduce a past formula  $\psi$  valid over a program (fair transition system)  $P$ , into a state formula  $\hat{\psi}$  which is valid over a program  $\hat{P}$ . Program  $\hat{P}$  is obtained by augmenting  $P$  with auxiliary boolean variables corresponding to the subformulas of  $\psi$ . Since  $\hat{\psi}$  is a state formula, we use the completeness result for state invariants (Section 2.5) to construct a proof of  $\hat{P} \models \hat{\psi}$ . We then show how to transform this proof into a proof of  $P \models \psi$  using rules and axioms of our temporal proof system.

### Elimination of Past Subformulas (“Statification”)

Let  $\Phi_\psi$  be the set of subformulas of  $\psi$  (possibly including  $\psi$  itself) whose principal operator is a past operator, e.g.,  $\ominus$  or  $\mathcal{S}$ .

**Example** If  $\psi$  is the formula

$$\psi: x = 10 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3)),$$

then  $\Phi_\psi$  is the set of formulas

$$\begin{aligned}\varphi_1 &: \Diamond(x = 3) \\ \varphi_2 &: \Diamond(x = 5 \wedge \Diamond(x = 3)).\end{aligned}$$

We refer to  $\Phi_\psi$  as the set of *principally-past subformulas* (PPS) of  $\psi$ . When  $\psi$  is understood from the context we may refer to  $\Phi_\psi$  simply as  $\Phi$ .

We define a set of new boolean variables  $B_\psi$  consisting of a variable  $b_\varphi$  for each formula  $\varphi \in \Phi_\psi$ . For the previously considered  $\psi$ ,  $B_\psi$  consists of  $b_{\varphi_1}$  and  $b_{\varphi_2}$ . When there is no danger of confusion, we refer to  $b_{\varphi_i}$  simply as  $b_i$ .

Let  $q$  be a subformula of  $\psi$ . We define the *statification* of  $q$ , denoted by  $stat(q)$ , inductively as follows:

- If  $q$  is a state formula, then  $stat(q) = q$ .
- If  $q$  is a PPS, i.e.,  $q \in \Phi_\psi$ , then  $stat(q) = b_q$ .
- If  $q$  has the form of a non-past operator  $\otimes$  applied to one or more subformulas, i.e.,  $q = \otimes(q_1, \dots, q_m)$ , then

$$stat(q) = \otimes(stat(q_1), \dots, stat(q_m)).$$

Essentially,  $stat(q)$  is obtained by replacing each  $\varphi$ , a PPS of  $q$ , by its corresponding boolean variable  $b_\varphi$ .

**Example** In the previous example, recalling that

$b_1$  encodes  $\varphi_1: \Diamond(x = 3)$ , and

$b_2$  encodes  $\varphi_2: \Diamond(x = 5 \wedge \Diamond(x = 3))$ ,

we obtain the following statifications for all the temporal subformulas of  $\psi$ :

$$\begin{aligned}stat(\Diamond(x = 3)) &= b_1 \\ stat(x = 5 \wedge \Diamond(x = 3)) &= (x = 5 \wedge b_1) \\ stat(\Diamond(x = 5 \wedge \Diamond(x = 3))) &= b_2 \\ stat(\psi) &= (x = 10 \rightarrow b_2).\end{aligned}$$

## Transforming the Program

Transforming the past formula  $\psi$  into a state formula  $stat(\psi)$  by the statification transformation is not sufficient. To ensure full correspondence between past formulas and their statifications, it is necessary that at all positions of the computation, the boolean variable  $b_{\varphi_i}$  assumes the same truth-value as the past formula  $\varphi_i \in \Phi$  encoded by  $b_{\varphi_i}$ . To achieve this, we also transform the program  $P$ , given by the fair transition system  $\langle V, T, \Theta, \mathcal{J}, \mathcal{C} \rangle$ , into its *statified version*  $\hat{P}_\psi$ , given

by  $(\widehat{V}, \widehat{T}, \widehat{\Theta}, \widehat{\mathcal{J}}, \widehat{\mathcal{C}})$ . The transformation is a special case of augmenting a program by auxiliary variables, a method introduced in Section 1.5.

We define

- $\widehat{V} = V \cup B_\psi$
- $\widehat{\Theta} = \Theta \wedge \bigwedge_{\varphi \in \Phi} (b_\varphi = (\varphi)_0)$ .

It is required that the initial value of  $b_\varphi$  equals the initial value of  $\varphi$ , i.e.  $(\varphi)_0$ , for each  $\varphi \in \Phi_\psi$ . Obviously,  $(\varphi)_0$  is a state formula referring only to the system variables  $V$ .

- $\widehat{T} = \{\widehat{\tau} \mid \tau \in T\}$ , where the transition relation  $\rho_{\widehat{\tau}}$  is given by

$$\rho_{\widehat{\tau}}: \rho_\tau \wedge \bigwedge_{\varphi \in \Phi} (b'_\varphi = \text{stat}(\varphi')).$$

Thus, the expression for the new value of each  $b_\varphi$ ,  $\varphi \in \Phi$ , is obtained by calculating  $\varphi'$  and then statifying the obtained formula. It is not difficult to see that  $\text{stat}(\varphi')$  is a state formula depending only on  $V$ ,  $V'$ , and  $B_\psi$ .

In the previous example, for  $\varphi_2$ :  $\Diamond(x = 5 \wedge \Diamond(x = 3))$ , we obtain

$$\begin{aligned} \varphi'_2 &= \varphi_2 \vee (x' = 5 \wedge (\Diamond(x = 3))') \\ &= \varphi_2 \vee (x' = 5 \wedge (\varphi_1 \vee x' = 3)), \end{aligned}$$

which can be simplified to

$$\varphi'_2 = \varphi_2 \vee (\varphi_1 \wedge x' = 5).$$

Therefore,

$$\text{stat}(\varphi'_2) = b_2 \vee (b_1 \wedge x' = 5).$$

Similarly

$$\text{stat}(\varphi'_1) = \text{stat}(\varphi_1 \vee x' = 3) = b_1 \vee x' = 3.$$

Thus, for each  $\tau \in T$ ,

$$\rho_{\widehat{\tau}}: \rho_\tau \wedge b'_1 = (b_1 \vee x' = 3) \wedge b'_2 = (b_2 \vee (b_1 \wedge x' = 5)).$$

- $\widehat{\mathcal{J}} = \{\widehat{\tau} \mid \tau \in \mathcal{J}\}$
- $\widehat{\mathcal{C}} = \{\widehat{\tau} \mid \tau \in \mathcal{C}\}$ .

**Example** Consider system INC of Fig. 4.2 (page 327), and the formula

$$\psi: x = 10 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3)).$$

As previously observed,

$$\Phi_\psi: \left\{ \underbrace{\Diamond(x = 3)}_{\varphi_1}, \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2} \right\}$$

$$B_\psi: \{b_{\varphi_1}, b_{\varphi_2}\},$$

which we represent as  $\{b_1, b_2\}$ .

The statified system  $\widehat{\text{INC}}_\psi$  is given by

- $\widehat{V}: \{x: \text{integer}; b_1, b_2: \text{boolean}\}$
- $\widehat{\Theta}: x = 0 \wedge b_1 = (x = 3) \wedge b_2 = (x = 5 \wedge x = 3),$   
which can be simplified to  

$$\widehat{\Theta}: x = 0 \wedge b_1 = b_2 = \text{F}.$$
- $\widehat{T}: \{\widehat{\tau}_I, \widehat{\tau}\}$  with transition relation (after simplifications)  

$$\rho_{\widehat{\tau}_I}: x' = x \wedge b'_1 = b_1 \wedge b'_2 = b_2$$
  

$$\rho_{\widehat{\tau}}: x' = x + 1 \wedge b'_1 = (b_1 \vee x' = 3) \wedge b'_2 = (b_2 \vee (b_1 \wedge x' = 5)).$$
- $\widehat{\mathcal{J}}: \{\widehat{\tau}\}$
- $\widehat{\mathcal{C}}: \{ \}.$  ■

## Computations of a Program and its Statified Version

Let  $\sigma: s_0, s_1, \dots$  be a computation of  $P$  and  $\widehat{\sigma}: \widehat{s}_0, \widehat{s}_1, \dots$  be the corresponding computation of  $\widehat{P}_\psi$ . Since  $\widehat{P}_\psi$  is an augmentation of  $P$ , there always exists such a corresponding computation, which is unique in its interpretation of  $\widehat{V} = V \cup B_\psi$ . It is obvious that the way the truth-value of  $\varphi \in \Phi$  changes from one state to its successor corresponds to the changes in the value of  $b_\varphi$ . Since the initial value of  $b_\varphi$  matches that of  $(\varphi)_0$ , we conclude that the value of  $\varphi$  at position  $j$  of  $\sigma$  is equal to the (boolean) value of  $b_\varphi$  at position  $j$  of  $\widehat{\sigma}$ . This is true for all subformulas of  $\psi$ , as is stated in the following claim:

**Claim 4.6** ( $\text{stat}(\varphi)$  mimics  $\varphi$ )

Let  $\sigma$  be a computation of  $P$  and  $\widehat{\sigma}$  the corresponding computation of  $\widehat{P}_\psi$ . For each  $\varphi$ , a subformula of  $\psi$ , and each position  $j = 0, 1, \dots$ , the value of  $\varphi$  at position  $j$  of  $\sigma$  equals the value of  $\text{stat}(\varphi)$  at position  $j$  of  $\widehat{\sigma}$ .

**Example** Consider the following computation of INC (Fig. 4.2, page 327):

$$\sigma: \langle x: 0 \rangle, \langle x: 1 \rangle, \langle x: 2 \rangle, \langle x: 3 \rangle, \langle x: 4 \rangle, \langle x: 5 \rangle, \langle x: 6 \rangle, \dots$$

and the corresponding computation of  $\widehat{\text{INC}}_\psi$ :

$$\widehat{\sigma}: \langle x: 0, b_1: F, b_2: F \rangle, \langle x: 1, b_1: F, b_2: F \rangle, \langle x: 2, b_1: F, b_2: F \rangle, \\ \langle x: 3, b_1: T, b_2: F \rangle, \langle x: 4, b_1: T, b_2: F \rangle, \langle x: 5, b_1: T, b_2: T \rangle, \\ \langle x: 6, b_1: T, b_2: T \rangle, \dots .$$

Obviously

$$\varphi_1: \Diamond(x = 3) \text{ holds at position } j \text{ of } \sigma \text{ iff } j \geq 3, \text{ and} \\ \varphi_2: \Diamond(x = 5 \wedge \Diamond(x = 3)) \text{ holds at position } j \text{ iff } j \geq 5.$$

The same is true of variables  $b_1$  and  $b_2$ , respectively. ■

## Using Rule INV

We now continue with the completeness proof. By assumption,  $\psi$  is an invariant of  $P$ , i.e.,  $\psi$  holds at all positions of all computations of  $P$ . According to the last claim,  $\widehat{\psi} = \text{stat}(\psi)$  holds at all positions of all computations of  $\widehat{P}_\psi$ , i.e., the state formula  $\widehat{\psi}$  is an invariant of  $\widehat{P}_\psi$ .

According to the completeness proof for state invariants (Section 2.5), there exists a state formula  $\widehat{\varphi}$  satisfying the premises of rule INV. That is, the following implications are state valid:

- I1.  $\widehat{\varphi} \rightarrow \widehat{\psi}$
- I2.  $\widehat{\Theta} \rightarrow \widehat{\varphi}$
- I3.  $\{\widehat{\varphi}\} \widehat{T} \{\widehat{\varphi}'\}$ .

We will use these implications to construct a proof of the invariance of formula  $\psi$  over program  $P$ . Examining the construction of  $\widehat{\varphi}$  described in Section 2.5, we observe that  $\widehat{\varphi}$  may refer to the auxiliary variables  $b_i \in B_\psi$ , but does not quantify over them.

**Example** Following the general procedure for system INC and formula

$$\psi: x = 10 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3)),$$

we obtained their stratified versions  $\widehat{\text{INC}}_\psi$ , and

$$\widehat{\psi} = \text{stat}(\psi): x = 10 \rightarrow b_2.$$

To prove the invariance of  $\widehat{\psi}$  over  $\widehat{\text{INC}}_\psi$ , we may use the auxiliary assertion

$$\widehat{\varphi}: (b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5).$$

It is not difficult to check that  $\widehat{\varphi}$  satisfies premises I1–I3 of rule INV with respect to  $\widehat{\text{INC}}_\psi$  and  $\widehat{\psi}$ . ■

## Unstatifying the Proof

We will show how to construct an auxiliary temporal formula  $\varphi$  and prove that it satisfies premises P1–P3 of rule INV-P, based on the first-order validities  $\widehat{f_1}$ – $\widehat{f_3}$ .

For a state formula  $\chi$  that may contain occurrences of variables  $b_i \in B_\psi$ , none of which is bound, we define its *unstatified version*, denoted by  $stat^{-1}(\chi)$ , as

$$stat^{-1}(\chi) = \chi[b_1 \mapsto \varphi_1, \dots, b_m \mapsto \varphi_m].$$

For example, for

$$\widehat{\varphi}: (b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5),$$

we have

$$stat^{-1}(\widehat{\varphi}) = (\Diamond(x = 3) \leftrightarrow x \geq 3) \wedge (\Diamond(x = 5 \wedge \Diamond(x = 3)) \leftrightarrow x \geq 5).$$

It is obvious that the transformation  $stat^{-1}$  is the inverse of  $stat$ . That is, for every  $\varphi$ , a subformula of  $\psi$ ,

$$stat^{-1}(stat(\varphi)) = \varphi.$$

In particular,

$$stat^{-1}(\widehat{\psi}) = \psi.$$

We take  $\varphi$  to be

$$\varphi: stat^{-1}(\widehat{\varphi})$$

and will prove each of the premises P1–P3 of rule INV-P.

The temporal proof rules that will be used are:

- *Instantiation* (rule INST):

$$\frac{\Vdash p \rightarrow q}{\models stat^{-1}(p) \rightarrow stat^{-1}(q)}$$

- *Instantiated generalization* (rule IGEN):

$$\frac{\Vdash p \rightarrow q}{\models stat^{-1}(p) \Rightarrow stat^{-1}(q)}$$

- *Temporalization* (rule TEMP):

$$\frac{\Vdash p \rightarrow q}{\models p \rightarrow q}$$

- *Entailment reasoning*:

$$\frac{\models \Box p \text{ and } \models p \Rightarrow q}{\models \Box q}$$

These rules can be derived from the deductive system for temporal logic presented in Chapter 3 of the SPEC book.

We will prove each of the premises P1–P3 in turn.

### Premise P1

From  $\widehat{I1}$ , we have

$$\Vdash \widehat{\varphi} \rightarrow \widehat{\psi}.$$

Applying rule IGEN, we obtain

$$\models \varphi \Rightarrow \psi,$$

which is premise P1 of rule INV-P.

**Example** For system  $\widehat{\text{INC}}_\psi$ , the intermediate assertion  $\varphi = \text{stat}^{-1}(\widehat{\varphi})$  is given by

$$\varphi: (\Diamond(x=3) \leftrightarrow x \geq 3) \wedge (\Diamond(x=5 \wedge \Diamond(x=3)) \leftrightarrow x \geq 5)$$

Formula  $\widehat{I1}$  for this example reads

$$\Vdash \underbrace{(b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5)}_{\widehat{\varphi}} \rightarrow \underbrace{x = 10 \rightarrow b_2}_{\widehat{\psi}},$$

which is obviously state valid.

Applying rule IGEN, we obtain

$$\models \varphi \Rightarrow \underbrace{x = 10 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3))}_{\psi}. \blacksquare$$

### Premise P2

From  $\widehat{I2}$ , we have

$$\Vdash \widehat{\Theta} \rightarrow \widehat{\varphi},$$

which can be written as

$$\Vdash \Theta \wedge \bigwedge_{\varphi \in \Phi} (b_\varphi = (\varphi)_0) \rightarrow \widehat{\varphi}.$$

Obviously, this implication is state valid iff the following implication is:

$$\Vdash \Theta \rightarrow \widehat{\varphi}[b_1 \mapsto (\varphi_1)_0, \dots, b_m \mapsto (\varphi_m)_0].$$

It can be shown that  $\widehat{\varphi}[b_1 \mapsto (\varphi_1)_0, \dots, b_m \mapsto (\varphi_m)_0]$  is equivalent to  $(\varphi)_0$ . Consequently, we obtain

$$\Vdash \Theta \rightarrow (\varphi)_0.$$

In **Problem 4.16**, we ask the reader to show that the unstatisfied formula  $\widehat{\varphi}[b_1 \mapsto (\varphi_1)_0, \dots, b_m \mapsto (\varphi_m)_0]$  is equivalent to  $(\varphi)_0$ .

**Example** For program  $\widehat{\text{INC}}_\psi$ , formula  $\widehat{I2}$  (in its unsimplified version) reads

$$\models \underbrace{x = 0 \wedge b_1 = (x = 3) \wedge b_2 = (x = 5 \wedge x = 3)}_{\widehat{\Theta}} \rightarrow \underbrace{(b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5)}_{\widehat{\varphi}},$$

which is valid, since the left-hand side implies

$$b_1 = b_2 = x \geq 3 = x \geq 5 = \text{F}.$$

Substituting for  $b_1, b_2$  on the right, we obtain

$$\models \underbrace{x = 0}_{\Theta} \rightarrow (x = 3 \leftrightarrow x \geq 3) \wedge (x = 5 \wedge x = 3 \leftrightarrow x \geq 5).$$

Observing that

$$x = 3 \text{ is } (\Diamond(x = 3))_0, \text{ and}$$

$$(x = 5 \wedge x = 3) \text{ is } (\Diamond(x = 5 \wedge \Diamond(x = 3)))_0,$$

the right-hand side of this implication is  $(\varphi)_0$ . ■

### Premise P3

For each  $\widehat{\tau} \in \widehat{T}$ , we have by  $\widehat{I3}$

$$\models \rho_\tau \wedge \underbrace{\bigwedge_{\varphi \in \Phi} (b'_\varphi = \text{stat}(\varphi'))}_{\widehat{\rho_\tau}} \wedge \widehat{\varphi} \rightarrow \widehat{\varphi}'.$$

This implication is state valid iff the following implication is:

$$\models \rho_\tau \wedge \widehat{\varphi} \rightarrow \widehat{\varphi}'[\dots, b'_i \mapsto \text{stat}(\varphi'_i), \dots].$$

Applying rule  $\text{IGEN}$  to this state validity, we obtain

$$\models \rho_\tau \wedge \varphi \Rightarrow \text{stat}^{-1}\left(\widehat{\varphi}'[\dots, b'_i \mapsto \text{stat}(\varphi'_i), \dots]\right).$$

A straightforward analysis of the definitions involved leads to the following claim of identities:

$$\text{stat}^{-1}\left(\widehat{\varphi}'[\dots, b'_i \mapsto \text{stat}(\varphi'_i), \dots]\right) = \widehat{\varphi}'[\dots, b'_i \mapsto \varphi'_i, \dots] = \varphi'.$$

Thus, we obtain premise P3 of  $\text{INV-P}$ :

$$\models \rho_\tau \wedge \varphi \Rightarrow \varphi'.$$

**Example** For system  $\widehat{\text{INC}}_\psi$ , formula  $\widehat{I3}$  reads

$$\models \underbrace{x' = x + 1 \wedge b'_1 = (b_1 \vee x' = 3) \wedge b'_2 = (b_2 \vee (b_1 \wedge x' = 5))}_{\rho_\tau} \wedge \\ \underbrace{(b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5)}_{\varphi} \rightarrow \underbrace{(b'_1 \leftrightarrow x' \geq 3) \wedge (b'_2 \leftrightarrow x' \geq 5)}_{\varphi'}$$

By case analysis, considering the different ranges of  $x$ , it can be ascertained that this implication is state valid.

Moving the equalities involving  $b'_1$  and  $b'_2$  to the right-hand side of the implication, we obtain

$$\models x' = x + 1 \wedge (b_1 \leftrightarrow x \geq 3) \wedge (b_2 \leftrightarrow x \geq 5) \rightarrow \\ (b_1 \vee x' = 3 \leftrightarrow x' \geq 3) \wedge (b_2 \vee (b_1 \wedge x' = 5) \leftrightarrow x' \geq 5)$$

The right-hand side formula is  $\varphi'$ :  $(b'_1 \leftrightarrow x' \geq 3) \wedge (b'_2 \leftrightarrow x' \geq 5)$  in which we have substituted the formula  $\text{stat}(\varphi'_1)$ :  $b_1 \vee x' = 3$  for  $b'_1$  and the formula  $\text{stat}(\varphi'_2)$ :  $b_2 \vee (b_1 \wedge x' = 5)$  for  $b'_2$ .

Applying rule IGEN to both sides, we obtain

$$\models x' = x + 1 \wedge \left( \underbrace{(\Diamond(x = 3) \leftrightarrow x \geq 3)}_{\varphi_1} \right) \wedge \\ \left( \underbrace{(\Diamond(x = 5 \wedge \Diamond(x = 3)) \leftrightarrow x \geq 5)}_{\varphi_2} \right) \Rightarrow \left( \underbrace{(\Diamond(x = 3) \vee (x' = 3) \leftrightarrow x' \geq 3)}_{(\Diamond x=3)'} \right) \\ \wedge \left( \underbrace{(\Diamond(x = 5 \wedge \Diamond(x = 3)) \wedge (x' = 5 \wedge \Diamond(x = 3))) \leftrightarrow x' \geq 5}_{(\Diamond(x=5 \wedge \Diamond(x=3)))'} \right)$$

It is straightforward to identify this entailment as the verification condition

$$\models \underbrace{x' = x + 1}_{\rho_\tau} \wedge \varphi \Rightarrow \varphi'. \blacksquare$$

## Conclusion

The proof of completeness establishes that rule INV-P is adequate for proving all safety formulas that are  $P$ -valid.

The presented completeness proof not only establishes this theoretical result, but also suggests a proof approach that is often used in practice: that of

statification. When the past formula  $\psi$  becomes too complex, such as the order-preservation property studied for program PROD-CONS (Fig. 4.3),

$$\psi: \text{cons}(u_2) \wedge \underbrace{\Diamond \text{cons}(u_1)}_{b_1} \rightarrow \underbrace{\Diamond (\text{prod}(u_2) \wedge \underbrace{\Diamond \text{prod}(u_1)}_{b_2})}_{b_3},$$

we may apply statification to it, obtaining the state formula

$$\hat{\psi}: \text{cons}(u_2) \wedge b_1 \rightarrow b_3,$$

where boolean variables  $b_1$ ,  $b_2$ , and  $b_3$  encode the denoted past formulas.

The state formula  $\hat{\psi}$  may be verified over the statified version  $\widehat{\text{PROD-CONS}}_\psi$  of program PROD-CONS presented in Fig. 4.16. In **Problem 4.17**, we request the reader to prove the invariance of  $\hat{\psi}$  over  $\widehat{\text{PROD-CONS}}_\psi$ .

Note that several optimization steps have been taken in the presentation of this program. Variables  $b_1$ ,  $b_2$ , and  $b_3$  are updated only where they may change. Secondly, using the sequential structure of the grouped statements, the reference to  $x$  in the statement updating  $b_2$  is actually to  $x'$ , the new value of  $x$ . In a similar way, the expression updating  $b_3$  refers to  $b'_2$ , the recently updated version of  $b_2$ .

## \* 4.10 Finite-State Algorithmic Verification

In this section we present an algorithm for checking that a safety formula  $\square \psi$  holds over a finite-state program  $P$ .

As examples for the algorithm, we use two systems, system INC1 (Fig. 4.17) and system INC2 (Fig. 4.18). The transition relations of both systems use addition modulo 8, represented as  $\oplus_8$ . System INC2 differs from INC1 by having an additional transition that increments  $x$  by 2, modulo 8.

The property we wish to check for these two systems is

$$\square \left( \underbrace{x = 7 \rightarrow \Diamond (x = 5 \wedge \Diamond (x = 3))}_{\psi_7} \right).$$

### Splitting States

For the simple case that  $\psi$  is a state formula, Section 2.6 suggests the following simple-minded approach: construct  $G_P$  the state-transition graph for program  $P$ , which contains all the accessible states of  $P$ , and evaluate  $\psi$  over each of the states. If all states in  $G_P$  satisfy  $\psi$  then  $\square \psi$  is  $P$ -valid. Otherwise  $\square \psi$  is not  $P$ -valid.

This approach does not work in general for an arbitrary past formula  $\psi$ .

in  $u_1, u_2 : \text{integer}$   
 local  $send, ack : \text{channel } [1..] \text{ of integer}$   
     where  $send = \Lambda, ack = \underbrace{[1, \dots, 1]}_N$   
 $b_1, b_2, b_3 : \text{boolean where } b_1 = b_2 = b_3 = F$

$Prod :: \left[ \begin{array}{l} \text{local } x, t : \text{integer} \\ \ell_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} \text{produce } x \\ \ell_1: \left\langle b_2 := b_2 \vee (x = u_1) \atop b_3 := b_3 \vee (x = u_2 \wedge b_2) \right\rangle \\ \ell_2: ack \Rightarrow t \\ \ell_3: send \Leftarrow x \end{array} \right] \end{array} \right]$

||

$Cons :: \left[ \begin{array}{l} \text{local } y : \text{integer} \\ m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: send \Rightarrow y \\ m_2: \left\langle ack \Leftarrow 1 \atop b_1 := b_1 \vee (y = u_1) \right\rangle \\ m_3: consume y \end{array} \right] \end{array} \right]$

Fig. 4.16. Program  $\widehat{\text{PROD-CONS}}_{\psi}$  (statified version of PROD-CONS).

$V: \{x \text{ integer: } [0..7]\}$   
 $\Theta: x = 0$   
 $T: \{\tau_I, \tau\} \text{ where } \rho_{\tau}: x' = x \oplus_8 1$   
 $\mathcal{J} = \mathcal{C}: \{ \}$

Fig. 4.17. System INC1.

$V$ : { $x$  integer: [0..7]}

$\Theta$ :  $x = 0$

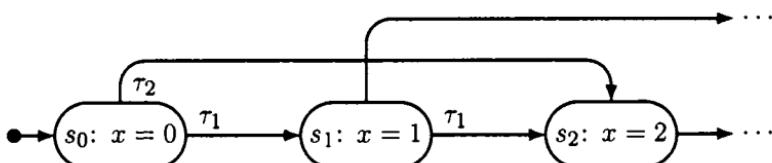
$\mathcal{T}$ :  $\{\tau_I, \tau_1, \tau_2\}$  where  $\rho_{\tau_1}$ :  $x' = x \oplus_8 1$   
 $\rho_{\tau_2}$ :  $x' = x \oplus_8 2$

$\mathcal{J} = \mathcal{C}$ : {}

Fig. 4.18. System INC2.

Consider Fig. 4.19, which contains a fragment of the state-transition graph of system INC2.

Consider the past formula  $\psi$ :  $\Diamond(x = 1)$ . We cannot evaluate this formula correctly on state  $s_2$ . This is because  $\psi$  not only depends on the interpretation of  $x$  at  $s_2$ , but also on the history that led to  $s_2$ . Here,  $s_2$  can be reached by the computation segment

Fig. 4.19. A fragment of  $G_{INC2}$ .

$$s_0: \langle x: 0 \rangle \xrightarrow{\tau_1} s_1: \langle x: 1 \rangle \xrightarrow{\tau_1} s_2: \langle x: 2 \rangle$$

which satisfies  $\Diamond(x = 1)$  at  $s_2$ , but also by the computation segment

$$s_0: \langle x: 0 \rangle \xrightarrow{\tau_2} s_2: \langle x: 2 \rangle,$$

which does not satisfy  $\Diamond(x = 1)$  at  $s_2$ .

The solution to this difficulty is that we must keep two copies of  $s_2$ . One copy corresponds to the case that  $s_2$  is reachable by a segment satisfying  $\psi$ , and this copy can be labeled by  $\psi$ . The other copy corresponds to segments reaching  $s_2$  which do not satisfy  $\psi$ , and this copy can be labeled by  $\neg\psi$ .

The algorithm presented here for arbitrary past formulas  $\psi$  is based on this idea of splitting states into several copies according to the relevant past formulas that hold on each of the copies.

## Atoms

For the rest of the section we assume a fixed finite-state program  $P$  and a past formula  $\psi$ , wishing to check whether  $\square \psi$  is  $P$ -valid.

Following the definition of Section 4.9, we construct  $\Phi_\psi$ , the set of all principally past subformulas (PPS) of  $\psi$ .

For example, the PPS of

$$\psi_7: x = 7 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3))$$

is given by

$$\Phi_{\psi_7}: \left\{ \underbrace{\Diamond(x = 3)}_{\varphi_1}, \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2} \right\}.$$

A  $(P, \psi)$ -atom is an interpretation of all the system variables in  $V$  and all the formulas in  $\Phi_\psi$ , assigning to each  $y \in V$  a value in its domain and to each  $\varphi \in \Phi_\psi$  a truth-value. We represent this interpretation by a list that includes, for each system variable its assigned value and for each formula  $\varphi \in \Phi_\psi$ , either  $\varphi$  itself (implying  $\varphi = T$ ) or its negation (implying  $\varphi = F$ ).

For example,

$$\alpha: \langle x: 7; \neg \Diamond(x = 3), \Diamond(x = 5 \wedge \Diamond(x = 3)) \rangle$$

is an  $(INC1, \psi_7)$ -atom which assigns to  $x$  the value 7 and to formulas  $\varphi_1: \Diamond(x = 3)$  and  $\varphi_2: \Diamond(x = 5 \wedge \Diamond(x = 3))$  the truth-values  $F$  and  $T$ , respectively.

Note that an atom can be viewed as a state labeled by positive and negative occurrences of the formulas in  $\Phi_\psi$ .

## Reachable-Atoms Graph

Instead of constructing the state-transition graph  $G_P$ , as we did for the case that  $\psi$  is a state formula, we now construct the *reachable-atoms graph*  $\mathcal{A}_{(P, \psi)}$  (abbreviated to  $\mathcal{A}$  whenever possible).

For a formula  $\varphi$  and a boolean value  $b$ , we define the notation

$$\varphi^b: \begin{cases} \varphi & \text{if } b = T \\ \neg\varphi & \text{if } b = F. \end{cases}$$

With this notation, an atom can be presented as

$$\alpha: \langle s; \varphi_1^{b_1}, \dots, \varphi_n^{b_n} \rangle,$$

where  $s$  is a state (interpreting  $V$ ), and  $\varphi_1^{b_1}, \dots, \varphi_n^{b_n}$  are positive or negative occurrences of all the formulas in  $\Phi_\psi$ .

For a state  $s$  and a state formula  $p$ , we denote by  $s[p]$  the boolean value resulting from evaluating  $p$  over  $s$ .

### Initial Atoms

Let  $s$  be an initial state, i.e., a state satisfying  $s \models \Theta$ . Such a state induces a unique atom  $\alpha_s$  given by

$$\alpha_s: \langle s; \varphi_1^{s[(\varphi_1)_0]}, \dots, \varphi_n^{s[(\varphi_n)_0]} \rangle$$

That is, for each  $\varphi_i \in \Phi_\psi$ , we evaluate  $(\varphi_i)_0$ , the initial value of  $\varphi_i$ , over  $s$ . If  $s[(\varphi_i)_0] = \top$  then  $\alpha_s$  interprets  $\varphi_i$  as  $\top$  (i.e., includes the entry  $\varphi_i$ ). Otherwise,  $\alpha_s$  interprets  $\varphi_i$  as  $\perp$  (i.e., includes the entry  $\neg\varphi_i$ ).

**Example** The initial state of `INC1` is  $s_0: \langle x: 0 \rangle$ . Evaluating

$$(\varphi_1)_0 = (\Diamond(x = 3))_0 = (x = 3)$$

$$(\varphi_2)_0 = (\Diamond(x = 5 \wedge \Diamond(x = 3)))_0 = (x = 5 \wedge x = 3)$$

over  $s_0$ , we obtain  $\perp$  in both cases. Consequently, the only initial atom in  $\mathcal{A}_{(\text{INC1}, \psi_7)}$  is

$$\alpha_0: \langle x: 0; \neg\Diamond(x = 3), \neg\Diamond(x = 5 \wedge \Diamond(x = 3)) \rangle \blacksquare$$

The rationale for this construction is provided by the following trivial claim:

#### Claim 4.7 (initial atoms)

Let  $\alpha: \langle s_0; \varphi_1^{b_1}, \dots, \varphi_n^{b_n} \rangle$  be an initial atom constructed according to the described prescription, and  $\sigma: s_0, s_1, \dots$  be a computation starting at  $s_0$ . Then, for each  $\varphi_i \in \Phi_\psi$ , the value of  $\varphi_i$  at position 0 in  $\sigma$  is precisely  $b_i$ .

Thus, the truth-value assigned to each  $\varphi_i$  in  $\alpha$  is precisely the value  $\varphi_i$  has at position 0 of all computations whose first state is  $s_0$ .

### Successor Atoms

Let  $\alpha: \langle s; \varphi_1^{b_1}, \dots, \varphi_n^{b_n} \rangle$  be an atom contained in  $\mathcal{A}$ , and let  $s'$  be a  $\tau$ -successor of  $s$  for some transition  $\tau$ . We intend to construct an atom  $\alpha': \langle s'; \varphi_1^{b'_1}, \dots, \varphi_n^{b'_n} \rangle$  that can serve as a successor to  $\alpha$ .

For each  $\varphi_i \in \Phi_\psi$ , compute the formula  $\varphi'_i$ . This formula refers at most to old variables  $y \in V$ , to new (primed) variables  $y' \in V'$ , and to old past formulas  $\varphi_j \in \Phi_\psi$ . Consequently, we can evaluate  $\varphi'_i$  over the joint interpretation  $\langle \alpha, s' \rangle$ , where  $y \in V$  is interpreted as  $s[y]$ ,  $y' \in V'$  is interpreted as  $s'[y]$ , and old formulas  $\varphi_j \in \Phi_\psi$  are interpreted as  $\alpha[\varphi_j] = b_j$ . Denote by  $\langle \alpha, s' \rangle[\varphi'_i]$  the truth-value obtained by this evaluation.

Consequently, we construct the atom  $\alpha'$ :  $\langle s'; \varphi_1^{b'_1}, \dots, \varphi_n^{b'_n} \rangle$ , where, for each  $\varphi_i \in \Phi_\psi$ ,  $b'_i = \langle \alpha, s' \rangle[\varphi'_i]$ . We refer to  $\alpha'$  as a *successor atom* of  $\alpha$ . Note that  $\alpha'$  is uniquely determined by  $\alpha$  and the successor state  $s'$ .

**Example** Assume that we have already included in the reachable-atoms graph for system INC1 the  $(\text{INC1}, \psi_7)$ -atom

$$\alpha_4: \quad \left\langle x: 4; \Diamond(x = 3), \neg \Diamond(x = 5 \wedge \Diamond(x = 3)) \right\rangle.$$

State  $s_4$ :  $\langle x: 4 \rangle$  has the successor  $s_5$ :  $\langle x: 5 \rangle$ . Computing the expressions  $\varphi'_i$  for  $i = 1, 2$ , yields (after simplifications)

$$\varphi'_1 = (\Diamond(x = 3))' = (x' = 3) \vee \underbrace{\Diamond(x = 3)}_{\varphi_1}$$

$$\begin{aligned} \varphi'_2 &= (\Diamond(x = 5 \wedge \Diamond(x = 3)))' \\ &= \left( x' = 5 \wedge (x' = 3 \vee \underbrace{\Diamond(x = 3)}_{\varphi_1}) \right)' \vee \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2}. \end{aligned}$$

Evaluating these two expressions over the joint interpretation

$$\langle \alpha_4, s' \rangle: \quad \left\langle x: 4, \Diamond(x = 3), \neg \Diamond(x = 5 \wedge \Diamond(x = 3)), x': 5 \right\rangle,$$

we obtain

$$\langle \alpha_4, s' \rangle[\varphi'_1] = F \vee T = T$$

$$\langle \alpha_4, s' \rangle[\varphi'_2] = (T \wedge (F \vee T)) \vee F = T.$$

Consequently, we obtain the following successor atom:

$$\alpha': \quad \left\langle x: 5; \Diamond(x = 3), \Diamond(x = 5 \wedge \Diamond(x = 3)) \right\rangle. \blacksquare$$

The rationale for this construction is provided by the following claim:

#### Claim 4.8 (successor atoms)

Let  $\alpha': \langle s_{k+1}; \varphi_1^{b'_1}, \dots, \varphi_n^{b'_n} \rangle$  be a successor of  $\alpha: \langle s_k; \varphi_1^{b_1}, \dots, \varphi_n^{b_n} \rangle$ , constructed according to the described prescription, and let  $\sigma: s_0, s_1, \dots$  be a computation, containing  $s_{k+1}$  as the successor of  $s_k$ . Assume that, for each  $\varphi_i \in \Phi_\psi$ ,  $(\sigma, k) \models \varphi_i^{b_i}$ . Then, for each  $\varphi_i \in \Phi_\psi$ ,  $(\sigma, k + 1) \models \varphi_i^{b'_i}$ .

The claim states that if  $\alpha$  is a valid description of the situation at position  $k$ , including the truth-values of the formulas  $\varphi_i \in \Phi_\psi$ , then  $\alpha'$  is a valid description of the situation at position  $k + 1$ .

## Construction of $\mathcal{A}$

The following algorithm constructs the reachable-atoms graph  $\mathcal{A}_{(P,\psi)}$ :

**Algorithm GRAPH** — constructing  $\mathcal{A}_{(P,\psi)}$

To construct the reachable-atoms graph  $\mathcal{A}_{(P,\psi)}$ , perform the following steps:

*Base:* Construct all the initial atoms and place them as nodes of  $\mathcal{A}$ . No edges are drawn. All these nodes are considered unmarked.

*Iteration:* Repeat the following until all nodes are marked. For each unmarked atom  $\alpha \in \mathcal{A}$ , let  $\alpha_1, \dots, \alpha_k$  be its successors.

- For each  $i = 1, \dots, k$ , if  $\alpha_i$  is not already in  $\mathcal{A}$ , add it to  $\mathcal{A}$  as an unmarked node.
- For each  $i = 1, \dots, k$ , draw an edge, labeled by the appropriate transition from  $\alpha$  to  $\alpha_i$ .
- Mark  $\alpha$ .

**Example** Applying the algorithm to system INC1 while ignoring the idling transition, we obtain the reachable-atoms graph presented in Fig. 4.20. As we see, there are two atoms for each state  $\langle x: j \rangle$  for  $j = 0, \dots, 4$ . Atom  $\alpha_j$  has  $x = j$  and  $\varphi_2$  false while atom  $\beta_j$  has  $x = j$  but  $\varphi_2$  true. The curved arrow entering  $\alpha_0$  identifies it as an initial atom. Full consideration of the idling transition would have added  $\tau_I$ -labeled edges, connecting each atom to itself. ■

## Properties of the Constructed Graph

Let  $\sigma: s_0, s_1, \dots$  be a computation of  $P$ . For each position  $j \geq 0$  we can construct an atom  $\alpha_j: \langle s_j; \varphi_1^{b_1}, \dots, \varphi_n^{b_n} \rangle$ , where  $b_i$  is the truth-value of  $\varphi_i$  at position  $j$ . We say that the sequence of these atoms  $a_\sigma: \alpha_0, \alpha_1, \dots$  is the *atom sequence induced* by the computation  $\sigma$ .

For example, the atom sequence

$$\langle x: 0; \neg\varphi_1, \neg\varphi_2 \rangle, \langle x: 1; \neg\varphi_1, \neg\varphi_2 \rangle, \langle x: 2; \neg\varphi_1, \neg\varphi_2 \rangle, \langle x: 3; \varphi_1, \neg\varphi_2 \rangle, \dots$$

is induced by the following computation of INC1:

$$\langle x: 0 \rangle, \langle x: 1 \rangle, \langle x: 2 \rangle, \langle x: 3 \rangle, \dots$$

An atom sequence  $\alpha_0, \alpha_1, \dots$  is said to be an *initialized path* in  $\mathcal{A}$  if  $\alpha_0$  is an initial atom and  $\alpha_{i+1}$  is a successor (atom) of  $\alpha_i$ , for every  $i = 0, 1, \dots$ .

An atom is called ( $P$ -)reachable if it appears in some atom sequence induced by a computation of  $P$ . The following claim states that the graph  $\mathcal{A}$  contains precisely the reachable atoms of  $P$ :

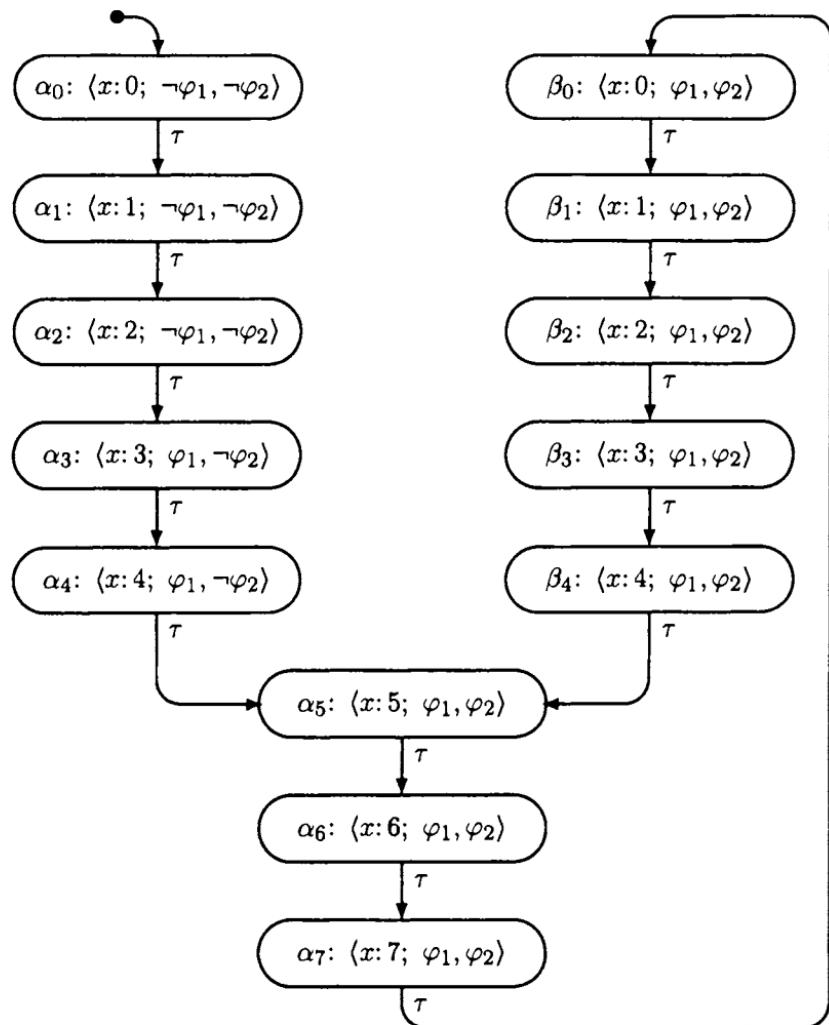


Fig. 4.20. Reachable-atoms graph for INC1.

**Claim ( $\mathcal{A}$  are the reachable atoms)**

A  $(P, \psi)$ -atom is  $P$ -reachable iff it appears in  $\mathcal{A}_{(P, \psi)}$ .

The proof of this claim is based on showing a one-to-one correspondence between computations of  $P$  and initialized infinite paths in  $\mathcal{A}$ . We show that every atom sequence induced by a computation of  $P$  is an initialized infinite path in  $\mathcal{A}$ . In the other direction, every initialized infinite path  $(s_0; \bar{\varphi}^{b_0}), (s_1; \bar{\varphi}^{b_1}), \dots$  in  $\mathcal{A}$  is an atom sequence induced by the state sequence  $s_0, s_1, \dots$ , which is a computation of  $P$ .

## Checking the $P$ -Invariance of $\psi$

The notion of  $(P, \psi)$ -atoms provides an explicit interpretation only for the subformulas of  $\psi$  whose principal operator is a past operator, i.e., to formulas that belong to  $\Phi_\psi$ . However, it is straightforward to extend this interpretation to any other subformula of  $\psi$ . This is because each  $p$ , a subformula of  $\psi$ , is a boolean combination of some formulas in  $\Phi_\psi$  and some state formulas.

**Example** The subformulas of

$$\psi_7: x = 7 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3))$$

and their expression as a boolean combinations of formulas in

$$\Phi_{\psi_7}: \left\{ \underbrace{\Diamond(x = 3)}_{\varphi_1}, \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2} \right\}$$

are

$$x = 7, x = 5, x = 3, \quad (\text{state formulas})$$

$$\underbrace{\Diamond(x = 3)}_{\varphi_1}, \quad \underbrace{x = 5 \wedge \Diamond(x = 3)}_{x=5 \wedge \varphi_1}, \quad \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2},$$

and

$$\underbrace{x = 7 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3))}_{x=7 \rightarrow \varphi_2}.$$

Consequently, given an atom  $\alpha$ , we can evaluate any subformula of  $\psi$ , including  $\psi$  itself, over  $\alpha$ . Denote by  $\alpha[\psi]$  the truth-value obtained by evaluating  $\psi$  over  $\alpha$ .

Consider the case that  $\psi$  is not an invariant of  $P$ . In this case, there exists a computation  $\sigma: s_0, s_1, \dots$  and a position  $j \geq 0$  such that  $(\sigma, j) \models \neg\psi$ . If we consider the atom sequence  $a_\sigma: \alpha_0, \alpha_1, \dots$  induced by  $\sigma$ , it follows that  $\psi$  evaluates to  $F$  over  $\alpha_j$ . Thus, if  $\psi$  is not a  $P$ -invariant, there exists a reachable atom,  $\alpha_j$ , such that  $\alpha_j[\psi] = F$ . It is not difficult to see that the other direction of the implication also holds: if  $\alpha[\psi] = T$  for some reachable atom  $\alpha$ , then  $\psi$  is not a  $P$ -invariant. This analysis and its consequences are summarized by the following claim:

### Claim 4.9 ( $P$ -invariance of $\psi$ )

The past formula  $\psi$  is a  $P$ -invariant iff  $\alpha[\psi] = T$  for all atoms  $\alpha \in \mathcal{A}_{(P, \psi)}$ .

This leads to the following algorithm for checking the validity of  $\Box \psi$  over the finite-state program  $P$ .

**Algorithm CHECK1** — checking for  $P$ -invariance

To check the  $P$ -invariance of  $\psi$ , i.e., whether  $P \models \square \psi$ , perform the following steps:

- Construct the reachable-atoms graph  $\mathcal{A}_{(P,\psi)}$ .
- If, for all  $\alpha \in \mathcal{A}_{(P,\psi)}$ ,  $\alpha[\psi] = T$ , report success:  $\square \psi$  is  $P$ -valid.
- Otherwise, report failure:  $\square \psi$  is not  $P$ -valid.

We illustrate the application of this algorithm by checking the validity of  $\square \psi_7$  over system INC1.

**Example** (INC1  $\models \square \psi_7$ )

The reachable-atoms graph for INC1 has been constructed in Fig. 4.20. It only remains to check whether

$$\psi_7: x = 7 \rightarrow \Diamond(x = 5 \wedge \Diamond(x = 3))$$

holds on all atoms appearing in this graph. Rewrite  $\psi_7$  as

$$x \neq 7 \vee \underbrace{\Diamond(x = 5 \wedge \Diamond(x = 3))}_{\varphi_2}.$$

We observe that all atoms except  $\alpha_7$  satisfy  $x \neq 7$ , and  $\alpha_7$  itself satisfies  $\varphi_2$ .

We conclude that INC1 satisfies  $\square \psi_7$ . ■

It is possible to improve the performance of the checking algorithm. Instead of first constructing the complete graph  $\mathcal{A}$  and only then checking whether all atoms satisfy  $\psi$ , we can interleave the two activities. As soon as a new atom is generated, we can check whether it satisfies  $\psi$ . If we encounter an atom  $\alpha$  such that  $\alpha[\psi] = F$ , we can stop immediately and declare that  $\psi$  is not a  $P$ -invariant. This leads to the following algorithm.

**Algorithm CHECK2** — on-the-fly checking for  $P$ -invariance

Follow the steps for the construction of the graph  $\mathcal{A}_{(P,\psi)}$ . Whenever a new atom  $\alpha$  is generated, compute  $\alpha[\psi]$ .

- If  $\alpha[\psi] = F$ , stop and report failure:  $\square \psi$  is not  $P$ -valid.
- If the construction terminated without encountering an atom falsifying  $\psi$ , report success:  $\square \psi$  is  $P$ -valid.

We illustrate the application of this improved version of the algorithm to system INC2.

**Example (INC2  $\not\models \square \psi_7$ )**

In Fig. 4.21 we present a fragment of the reachable-atoms graph. The construction is stopped as soon as atom  $\gamma_7$  or  $\delta_7$  is encountered. These atoms do not satisfy  $\psi_7$  as  $x = 7$  but  $\varphi_2 = F$ .

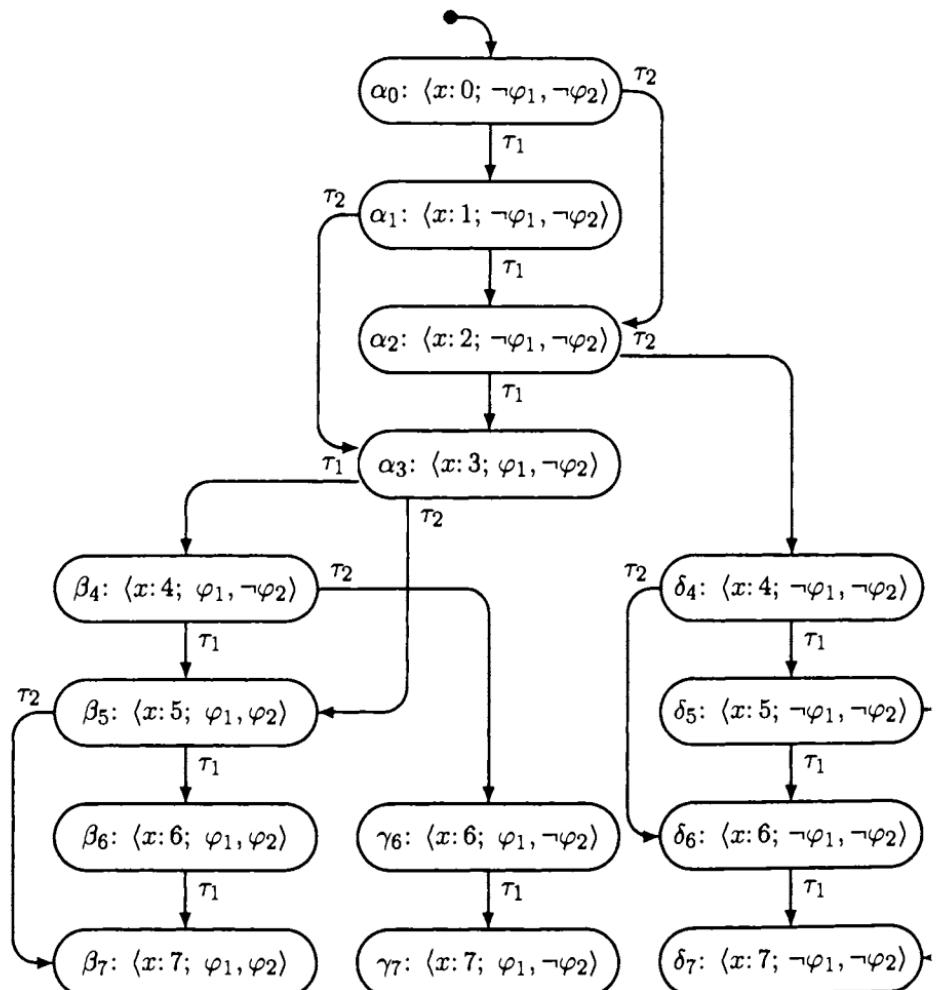


Fig. 4.21. A fragment of the reachable-atoms graph for INC2.

We conclude that INC2 does not satisfy  $\square \psi_7$ . ■

## Problems

**Problem 4.1** (invariance under stuttering) page 325

Let  $\varphi$  be a previous-free formula, i.e., a past formula that contains no occurrences of  $\ominus p$ ,  $\odot p$ , or  $x^-$ , contains no quantifiers, and refers only to system or rigid variables. Show by induction on the structure of  $\varphi$  that the verification condition

$$\rho_{\tau_I} \wedge \varphi \Rightarrow \varphi'$$

is  $P$ -valid.

**Problem 4.2** (every consumed value was previously produced) page 333

In Section 4.2, we presented a part of a proof that program PROD-CONS of Fig. 4.3 (page 331) satisfies the safety property

$$\psi: \text{cons}(u) \rightarrow \Diamond \text{prod}(u).$$

This property states that every value consumed must have been previously produced. Complete the proof of this property.

**Problem 4.3** (no-invented-replicates for data-independent programs) page 334

Prove Claim 4.3.

**Problem 4.4** (specifying no-invented-replicates with quantifiers) page 334

In the discussion about duplicate messages (page 333), we mentioned a theoretical result, by which the property of no-invented-replicates cannot be specified without using auxiliary variables or quantifiers.

(a) Write a temporal formula with quantifiers that specifies the property of no-invented-replicates. One possible approach is to use a quantified (flexible) variable  $m$  which ranges over *multisets* of messages. (A multiset is a set that allows multiple occurrences of elements). The formula may require that every  $\text{prod}(u)$  event (the production of message  $u$ ) adds  $u$  to the multiset  $n$ , and every  $\text{cons}(u)$  event (the consumption of message  $u$ ) removes one copy of  $u$  from  $m$ , and is possible only if  $m$  currently contains at least one copy of  $u$ .

(b) Write a quantifier-free formula  $\psi$  (which may use auxiliary variables such as  $m$  above) such that  $\psi$  is  $P$ -valid iff  $P$  has the no-invented-replicates property. You cannot assume that  $P$  is data independent. Recall that if  $m$  is an auxiliary variable not appearing in a program  $P$ , then  $\forall m: \varphi$  is  $P$ -valid iff  $\varphi$  is.

**Problem 4.5** (non-compositional verification of program KEEPING-UP) page 342

In Section 4.3, we presented a compositional proof of the invariance of the assertion

$$|x - y| \leq 1$$

over program KEEPING-UP of Fig. 4.6 (page 338). Reprove this invariant using the non-compositional methods of Chapters 1 and 2.

**Problem 4.6** (inclusive version of rule CAUS) page 345

In Fig. 4.22, we present rule CAUS-I, which can be viewed as an inclusive version of rule CAUS of Fig. 4.9 (page 343).

For past formulas  $p, r, \varphi$

$$\frac{\begin{array}{l} \text{I1. } p \Rightarrow \varphi \vee r \\ \text{I2. } \Theta \rightarrow \neg(\varphi)_0 \\ \text{I3. } \{\varphi \wedge \neg r\} T^{-1} \{\varphi \vee r\} \end{array}}{p \Rightarrow \Diamond r}$$

Fig. 4.22. Rule CAUS-I (inclusive causality).

(a) Is rule CAUS-I at least as powerful as rule CAUS? That is, can you construct a past formula  $\hat{\varphi}$  in terms of  $p, r$ , and  $\varphi$  such that, whenever past formulas  $p, r$ , and  $\varphi$  satisfy premises C1-C3 of rule CAUS, past formulas  $p, r$ , and  $\hat{\varphi}$  satisfy premises I1-I3 of rule CAUS-I?

(b) Is rule CAUS at least as powerful as rule CAUS-I?

**Problem 4.7** (different ways of expressing precedence) page 346

To express that (a state satisfying)  $p$  weakly precedes (a state satisfying)  $q$ , we may use the waiting-for formula  $(\neg q) W p$  or the past formula  $q \Rightarrow \Diamond p$ . Assume that  $p$  and  $q$  are assertions.

(a) Are formulas  $(\neg q) W p$  and  $q \Rightarrow \Diamond p$  equivalent?

(b) Are the two formulas congruent?

(c) Can every proof of  $(\neg q) W p$ , using rule WAIT, be transformed to a proof of  $q \Rightarrow \Diamond p$  by rule CAUS?

(d) Can every proof of  $q \Rightarrow \Diamond p$  be transformed to a proof of  $(\neg q) W p$ ?

**Problem 4.8** (order-preservation property) page 356

In Section 4.6, we stated the following order-preservation causality property for program PROD-CONS (Fig. 4.3, page 331):

$$\psi_2: \text{cons}(u_2) \wedge \Diamond \text{cons}(u_1) \Rightarrow \Diamond (\text{prod}(u_2) \wedge \Diamond \text{prod}(u_1)).$$

This property states that values are consumed in the same order that they are produced. The text presents part of the proof of this property, using rule CAUS and relying on two previously established invariants:

$$\begin{aligned}\psi_1: \text{cons}(u) &\Rightarrow \Diamond \text{prod}(u) \\ \psi'_1: u \in \text{send}^* &\Rightarrow \Diamond \text{prod}(u).\end{aligned}$$

The virtual variable  $\text{send}^*$  is defined as

$$\text{send}^* = (\text{if } at\_m_{2,3} \text{ then } (y) \text{ else } \Lambda) * \text{send} * (\text{if } at\_\ell_3 \text{ then } (x) \text{ else } \Lambda).$$

Present a complete proof of this property.

**Problem 4.9** (order replication for data-independent programs) page 357

Formulate and prove a claim similar to Claim 4.3 which states that formula

$$\psi_2: \text{cons}(u_2) \wedge \Diamond \text{cons}(u_1) \Rightarrow \Diamond(\text{prod}(u_2) \wedge \Diamond \text{prod}(u_1))$$

specifies the property of order replication for data-independent programs.

**Problem 4.10** (quantified specification of order replication) page 357

(a) Write a temporal formula with quantifiers that specifies the property of order replication. One possible approach is to use a quantified (flexible) variable  $q$  which ranges over *queues* of messages. The formula may require that every  $\text{prod}(u)$  event (the production of message  $u$ ) appends  $u$  to the end of the queue  $q$ , and every  $\text{cons}(u)$  event (the consumption of message  $u$ ) removes one copy of  $u$  from the front of  $q$ , and is possible only if  $q$  has  $u$  at its front.

(b) Write a quantifier-free formula  $\psi$  (which may use auxiliary variables such as  $q$  above) such that  $\psi$  is  $P$ -valid iff  $P$  has the order-replication property. You cannot assume that  $P$  is data independent. Recall that if  $q$  is an auxiliary variable not appearing in a program  $P$ , then  $\forall q: \varphi$  is  $P$ -valid iff  $\varphi$  is.

**Problem 4.11** (proof by history variables) page 362

Prove the invariance of the following assertions over program PROD-CONS-H of Fig. 4.11 (page 358)

$$\begin{aligned}\chi_1: \Diamond \text{cons}(u) &\leftrightarrow u \in h_c \\ \chi_2: \Diamond \text{prod}(u) &\leftrightarrow u \in h_p \\ \chi_3: \Diamond(\text{cons}(u_2) \wedge \Diamond \text{cons}(u_1)) &\leftrightarrow \text{precede}(u_1, u_2, h_c) \\ \chi_4: \Diamond(\text{prod}(u_2) \wedge \Diamond \text{prod}(u_1)) &\leftrightarrow \text{precede}(u_1, u_2, h_p).\end{aligned}$$

The predicate  $\text{precede}(u_1, u_2, a)$  for integers  $u_1$  and  $u_2$  and integer list  $a$  is defined as

$$\text{precede}(u_1, u_2, a): \exists i, j: i \leq j: a[i] = u_1 \wedge a[j] = u_2.$$

**Problem 4.12** (nested waiting-for vs. back-to) page 364

Show the equivalence

$$p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} r \quad \sim \quad (\neg q_1) \Rightarrow q_2 \mathcal{B} q_3 \cdots q_m \mathcal{B} (\neg p) \mathcal{B} r,$$

by establishing

$$\begin{aligned} p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} r &\sim \quad \square((\neg p) \mathcal{W} q_m \cdots q_1 \mathcal{W} r), \\ p \Rightarrow q_m \mathcal{B} q_{m-1} \cdots q_1 \mathcal{B} r &\sim \quad \square((\neg p) \mathcal{B} q_m \cdots q_1 \mathcal{B} r), \end{aligned}$$

and

$$\square(q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} r) \sim \square(q_1 \mathcal{B} q_2 \cdots q_m \mathcal{B} r).$$

**Problem 4.13** (relation of rule NWAIT to rule NBACK) page 369

It is known (see Problem 4.12) that the following two formulas are equivalent:

$$\psi_1: \square(q_1 \mathcal{W} q_2 \cdots q_m \mathcal{W} r)$$

$$\psi_2: \square(q_m \mathcal{B} q_{m-1} \cdots q_1 \mathcal{B} r).$$

Show that a proof of  $\psi_1$  by rule NWAIT can be transformed to a proof of  $\psi_2$  by rule NBACK, and vice versa.

**Problem 4.14** (2-bounded overtaking) page 371

Prove a bound of 2 on the possible overtaking from  $\ell_3$  for program MUX-PET2 of Fig. 4.10 (page 350). That is, establish that the nested back-to formula

$$at\_m_5 \Rightarrow at\_m_5 \mathcal{B} (\neg at\_m_5) \mathcal{B} at\_m_5 \mathcal{B} \neg at\_\ell_{3,4}$$

is valid over MUX-PET2. In fact, this property has been described as  $1\frac{1}{2}$ -bounded overtaking in Section 3.3.

**Problem 4.15** (a nested back-to property of PROD-CONS) page 371

Prove the following nested back-to property of program PROD-CONS (Fig. 4.3, page 331):

$$y = u \Rightarrow (y = u) \mathcal{B} (u \in send) \mathcal{B} (x = u) \mathcal{B} prod(u).$$

**Problem 4.16** (unstatisfying the initial version of a formula) page 379

Let  $\varphi$  be a past formula, and  $\widehat{\varphi}$  be its statified version. Assume that the list of principally-past subformulas of  $\varphi$  is  $\varphi_1, \dots, \varphi_m$  and the list of their corresponding boolean variables is  $b_1, \dots, b_m$ . Let  $(\varphi)_0$  denote the initial version of  $\varphi$ , and let  $(\varphi_1)_0, \dots, (\varphi_m)_0$  denote the initial version of  $\varphi_1, \dots, \varphi_m$ . Show the following equivalence:

$$(\varphi)_0 \sim \widehat{\varphi}[b_1 \mapsto (\varphi_1)_0, \dots, b_m \mapsto (\varphi_m)_0].$$

**Problem 4.17** (proving order preservation by statification) page 381

Consider program  $\widehat{\text{PROD-CONS}}_\psi$  of Fig. 4.16 (page 382). This is a statified version of program PROD-CONS, obtained by augmentation with the following statification variables:

- $b_1$  — statifying  $\Diamond \text{cons}(u_1)$
- $b_2$  — statifying  $\Diamond \text{prod}(u_1)$
- $b_3$  — statifying  $\Diamond(\text{prod}(u_2) \wedge \Diamond \text{prod}(u_1))$ .

Prove the invariance of the state formula

$$\hat{\psi}: \text{cons}(u_2) \wedge b_1 \rightarrow b_3$$

over program  $\widehat{\text{PROD-CONS}}$ .

## Bibliographic Remarks

**Past operators:** As explained in the bibliographic notes of Chapter 0, the temporal systems studied in Logic and Philosophy, such as the system introduced in Kamp [1968], included both future and past operators. However, the first temporal systems considered in Computer Science, such as Pnueli [1977] and Manna and Pnueli [1981a], contained only the future fragment. The decision to add the past operators, as explained in Lichtenstein, Pnueli, and Zuck [1985], was motivated by the observation (Pnueli [1985]) that the past operators led to a cleaner approach to compositional verification, as presented in Section 4.3. Additional inspiration was provided by the two books, Smullyan [1992a, 1992b], which consider retrograde chess problems. These are chess problems in which the reader is presented with a board situation and asked how can one get into such a situation, or what could have been the preceding situations. To formalize and reason about such problems, using temporal logic, one obviously needs past operators.

**Duplicate messages:** The observation that a buffer with duplicate messages cannot be specified in propositional temporal logic was first made by Sistla, Clarke, Francez, and Gurevich [1982], and further elaborated by Sistla, Clarke, Francez, and Meyer [1984]. Some suggestions for extensions of temporal logic that will solve this difficulty are discussed by Koymans [1987]. The notion of *data independence* discussed in Section 4.7, which is one of the possible solutions to message duplication, is taken from Wolper [1986].

**Compositional verification:** Most of the proof approaches presented in this text can be described as global, in the sense that they require the complete program for establishing its properties. An alternative approach builds a proof of a property compositionally: first proving some simpler properties of a component

of the program, and then combining proven properties of the components in order to infer a property of the complete program. Section 4.3 presents such an approach. Our approach to compositional verification is a development of the ideas proposed in Barringer, Kuiper, and Pnueli [1984]. Additional compositional approaches to temporal verification were proposed in Nguyen, Demers, Owicki, and Gries [1986], Nguyen, Gries, and Owicki [1985], Jonsson [1987], Chandy and Misra [1988], and Abadi and Lamport [1993]. The literature abounds with non-temporal approaches to compositionality, some of which include the textbooks Apt and Olderog [1991] and Francez [1992], the surveys by de Roever [1985] and Hooman and de Roever [1986], and the monograph by Zwiers [1989]. Compositional approaches that include real time were studied in Chang [1994] and further developed in Chang, Manna, and Pnueli [1994].

**Completeness:** Most of the material in this chapter, including the *statification* approach and the proof of completeness, is based on Manna and Pnueli [1991a].

**Algorithmic verification:** The approach to the algorithmic verification of general safety properties, presented in Section 4.10, is a special case of the more general algorithmic verification procedures, presented in Chapter 5. Here, we only considered the special case of canonical safety formulas of the form  $\Box p$ , for a past formula  $p$ .



# Algorithmic Verification of General Formulas

At the end of each of Chapters 2–4, we presented an algorithm for checking whether a property belonging to the class of properties considered in that chapter is valid over a given finite-state system.

Thus, Section 2.6 presented an algorithm for finite-state verification of properties that can be specified by a formula of the form  $\Box p$ , for some assertion  $p$ . Section 3.6 presented a finite-state verification algorithm for properties specifiable by a formula of the form  $p \Rightarrow q_m \mathcal{W} q_{m-1} \cdots q_1 \mathcal{W} q_0$ , for assertions  $p, q_0, \dots, q_m$ . Finally, Section 4.10 presented a finite-state verification algorithm for a general safety property, specifiable by a formula of the form  $\Box p$ , for an arbitrary past formula  $p$ .

However, what does one do when the presented temporal formula  $\varphi$  is not in one of these canonical forms or easily transformable to such a form? This chapter provides the answer to this question, presenting a general algorithm for checking whether an arbitrary (state-quantified) formula  $\varphi$  is valid over a given finite-state system  $P$ .

Section 5.1 presents an algorithm for checking whether a given temporal formula  $\varphi$  is satisfiable in some model. The same algorithm can be used to check whether  $\varphi$  is valid. Section 5.2 sharpens the algorithm to check for satisfiability and validity of  $\varphi$  over computations of a given finite-state system  $P$ . This yields the desired algorithm for checking whether  $\varphi$  is  $P$ -valid. The problem of checking whether a temporal formula is  $P$ -valid over a finite-state system is often referred to as the *model-checking* problem, even though in the case of the linear-time temporal logic presented in this book, a finite-state system represents a set of models rather than a single model. Applications of this algorithm are illustrated in Section 5.3. The algorithms of Sections 5.1 and 5.2 are based on the construction of a *temporal tableau* for a formula  $\varphi$ . Section 5.4 describes an incremental construction of the tableau, which generates smaller tableaux. Section 5.5 improves the construction

further by considering only partial versions of the tableau.

Note that while Chapters 1–4 consider only safety properties, the algorithms presented in this chapter, being applicable to an arbitrary (state-quantified) temporal formula, can also be used for algorithmic verification of progress properties, which is the main topic of the PROGRESS book.

## 5.1 Satisfiability of a Temporal Formula

In this section, we present an algorithm for checking whether a temporal formula  $p$  is satisfiable. The algorithm will either answer positively, and produce an example model  $\sigma$  satisfying  $p$ , or answer negatively, implying that  $p$  is unsatisfiable. Applying the algorithm to the formula's negation  $\neg p$ , a positive answer means that  $\neg p$  is satisfiable, i.e.,  $p$  is not valid, and the produced model  $\sigma$  is a counterexample, while a negative answer implies that  $p$  is valid. Thus, the same algorithm can also be used to check for validity of  $p$ .

### The $\omega$ -Notation

For a sequence of elements  $e_1, \dots, e_k$ ,  $k \geq 1$ , we denote by

$$(e_1, e_2, \dots, e_k)^\omega$$

the infinite periodic sequence

$$e_1, \dots, e_k, e_1, \dots, e_k, \dots$$

When  $k = 1$ , we write  $(e)^\omega$  simply as  $e^\omega$ , which stands for  $e, e, \dots$

### Expansion Formulas

The requirement that a temporal formula holds at a position  $j$  of a model can often be decomposed into a requirement that a simpler formula holds at the same position and some other formula holds either at  $j + 1$  or at  $j - 1$ . We refer to this decomposition as an *expansion formula*. Following is a list of congruences presenting an expansion formula for each of the temporal operators, excluding  $\bigcirc$ ,  $\ominus$ , and  $\oslash$ .

$\Box p \approx p \wedge \bigcirc \Box p$	$\Xi p \approx p \wedge \ominus \Xi p$
$\Diamond p \approx p \vee \bigcirc \Diamond p$	$\Diamond p \approx p \vee \ominus \Diamond p$
$p \mathcal{U} q \approx q \vee [p \wedge \bigcirc(p \mathcal{U} q)]$	$p \mathcal{S} q \approx q \vee [p \wedge \ominus(p \mathcal{S} q)]$
$p \mathcal{W} q \approx q \vee [p \wedge \bigcirc(p \mathcal{W} q)]$	$p \mathcal{B} q \approx q \vee [p \wedge \ominus(p \mathcal{B} q)]$

## Closure and Atoms

We consider temporal formulas that use the constants  $T$  and  $F$ , general state formulas, boolean connectives  $\neg$ ,  $\vee$ , and  $\wedge$ , and the temporal operators  $\bigcirc$ ,  $\lozenge$ ,  $\Box$ ,  $\mathcal{U}$ ,  $\mathcal{W}$ ,  $\ominus$ ,  $\odot$ ,  $\boxdot$ ,  $\boxtimes$ ,  $\mathcal{S}$ , and  $\mathcal{B}$ . Quantifiers are allowed only at the level of state formulas.

Additional boolean connectives, such as  $\rightarrow$  and  $\leftrightarrow$ , can be expressed in terms of the previously listed basic operators, e.g., we can express  $p \rightarrow q$  as  $\neg p \vee q$  and  $p \leftrightarrow q$  as  $(p \rightarrow q) \wedge (q \rightarrow p)$ .

We define the *closure* of a formula  $\varphi$ , denoted by  $\Phi_\varphi$ , as the smallest set of formulas satisfying the following requirements:

- $\varphi \in \Phi_\varphi$ .
- For every  $p \in \Phi_\varphi$  and  $q$  a subformula of  $p$ ,  $q \in \Phi_\varphi$ .
- For every  $p \in \Phi_\varphi$ ,  $\neg p \in \Phi_\varphi$ .

To keep the closure finite, we identify  $\neg\neg p$  with  $p$  within any context.

- For every  $\psi \in \{\Box p, \lozenge p, p \mathcal{U} q, p \mathcal{W} q\}$ , if  $\psi \in \Phi_\varphi$  then  $\bigcirc \psi \in \Phi_\varphi$ .
- For every  $\psi \in \{\boxdot p, p \mathcal{S} q\}$ , if  $\psi \in \Phi_\varphi$  then  $\ominus \psi \in \Phi_\varphi$ .
- For every  $\psi \in \{\boxtimes p, p \mathcal{B} q\}$ , if  $\psi \in \Phi_\varphi$  then  $\odot \psi \in \Phi_\varphi$ .

Note the use of the *previous* operator  $\ominus$  for  $\boxdot$  and  $\mathcal{S}$  and the use of the *before* operator  $\odot$  for  $\boxtimes$  and  $\mathcal{B}$ .

The treatment of formulas in this chapter is mainly syntactic. For example, we consider the formula  $\neg \Box p$  to be distinct from the formula  $\lozenge \neg p$  and  $\bigcirc(p \vee p)$  to be distinct from  $\bigcirc(q \vee p)$ . There are few exceptions to this rule such as the identification of  $\neg\neg p$  with  $p$  in any context. Other exceptions will be explicitly identified whenever needed.

The formulas in  $\Phi_\varphi$  are called the *closure formulas* of  $\varphi$ .

**Example** The closure of the formula

$$\varphi_1: \quad \Box p \wedge \lozenge \neg p$$

is given by

$$\Phi_{\varphi_1}: \quad \left\{ \begin{array}{l} \varphi_1, \quad \Box p, \quad \lozenge \neg p, \quad \bigcirc \Box p, \quad \bigcirc \lozenge \neg p, \quad p \\ \neg \varphi_1, \quad \neg \Box p, \quad \neg \lozenge \neg p, \quad \neg \bigcirc \Box p, \quad \neg \bigcirc \lozenge \neg p, \quad \neg p \end{array} \right\}$$

and the closure of the formula

$$\varphi_2: \quad p \mathcal{U} \bigcirc q$$

is given by

$$\Phi_{\varphi_2}: \quad \left\{ \begin{array}{lllll} p \cup \bigcirc q, & \bigcirc(p \cup \bigcirc q), & \bigcirc q, & p, & q \\ \neg(p \cup \bigcirc q), & \neg \bigcirc(p \cup \bigcirc q), & \neg \bigcirc q, & \neg p, & \neg q \end{array} \right\}.$$

The closure  $\Phi_\varphi$  can be partitioned into two sets of equal size

$$\Phi_\varphi = \Phi_\varphi^+ \cup \Phi_\varphi^-,$$

where  $\Phi_\varphi^+$  contains all the closure formulas whose principal operator is not a negation, while  $\Phi_\varphi^-$  contains all the closure formulas whose principal operator is a negation. Thus, the closure of the formula  $\varphi_1: \square p \wedge \diamond \neg p$  can be partitioned into

$$\Phi_{\varphi_1}^+: \quad \{\varphi_1, \quad \square p, \quad \diamond \neg p, \quad \bigcirc \square p, \quad \bigcirc \diamond \neg p, \quad p\},$$

$$\Phi_{\varphi_1}^-: \quad \{\neg\varphi_1, \quad \neg \square p, \quad \neg \diamond \neg p, \quad \neg \bigcirc \square p, \quad \neg \bigcirc \diamond \neg p, \quad \neg p\}.$$

The size of the closure can be bounded by  $|\Phi_\varphi| \leq 4|\varphi|$ , where  $|\varphi|$  is the *size* of the formula  $\varphi$ , defined as the number of occurrences of boolean connectives, temporal operators, and atomic formulas. To see the reason for the constant 4, observe that a subformula of  $\varphi$  such as  $\square p$  contributes the following 4 elements to the closure:

$$\square p, \quad \neg \square p, \quad \bigcirc \square p, \quad \text{and} \quad \neg \bigcirc \square p.$$

Therefore, since a formula  $\varphi$  contains at most  $|\varphi|$  subformulas, and the closure is constructed from the individual contributions of these subformulas,  $|\Phi_\varphi| \leq 4|\varphi|$ .

For example, reconsidering formula  $\varphi_1$ , we have

$$|\Phi_{\varphi_1}| = 12 \leq 4 \cdot 6 = 4 \cdot |\varphi_1|.$$

## Classification of Formulas

We classify some of the temporal formulas into  $\alpha$ - and  $\beta$ -formulas according to the two tables presented in Fig. 5.1. A formula is called an  *$\alpha$ -formula* or a  *$\beta$ -formula* if it appears in the left column of the  $\alpha$ -table, or  $\beta$ -table, respectively.

For each  $\alpha$ -formula  $\varphi$ , the  $\alpha$ -table contains a set of formulas  $\kappa(\varphi)$ , which can be viewed as the consequences of  $\varphi$ . For each  $\beta$ -formula  $\psi$ , the  $\beta$ -table contains a formula  $\kappa_1(\psi)$  and a set of formulas  $\kappa_2(\psi)$ , which can be viewed as alternative consequences of  $\psi$ .

The intended meaning of a formula belonging to one of these tables is as follows. An  $\alpha$ -formula  $\varphi$  holds at position  $j$  iff all the  $\kappa(\varphi)$ -formulas hold at  $j$ . A  $\beta$ -formula  $\psi$  holds at position  $j$  iff either  $\kappa_1(\psi)$ , or all the  $\kappa_2(\psi)$ -formulas (or both) hold at  $j$ .

For example,  $\square p$  holds at position  $j$  in a model iff both  $p$  and  $\bigcirc \square p$  hold at  $j$ . On the other hand,  $p \cup q$  holds at position  $j$  iff either  $q$  holds at  $j$  or both  $p$  and  $\bigcirc(p \cup q)$  hold at  $j$ .

$\alpha$	$\kappa(\alpha)$	$\beta$	$\kappa_1(\beta)$	$\kappa_2(\beta)$
$p \wedge q$	$p, q$	$p \vee q$	$p$	$q$
$\square p$	$p, \bigcirc \square p$	$\Diamond p$	$p$	$\bigcirc \Diamond p$
$\exists p$	$p, \ominus \exists p$	$\Diamond \exists p$	$p$	$\ominus \Diamond p$
		$p \mathcal{U} q$	$q$	$p, \bigcirc(p \mathcal{U} q)$
		$p \mathcal{W} q$	$q$	$p, \bigcirc(p \mathcal{W} q)$
		$p \mathcal{S} q$	$q$	$p, \ominus(p \mathcal{S} q)$
		$p \mathcal{B} q$	$q$	$p, \ominus(p \mathcal{B} q)$

Fig. 5.1.  $\alpha$ - and  $\beta$ -tables.

The tables of Fig. 5.1 are based on the expansion formulas presented in page 400. Notice that the tables of Fig. 5.1 do not include entries for the operators  $\bigcirc$  or  $\ominus$ . This is because formulas  $\bigcirc p$  or  $\ominus p$  holding at position  $j$  do not imply anything about their subformulas holding at the same position. They may imply that some of their subformulas should hold at other positions, but this is not expressed in the tables of Fig. 5.1.

## Atoms

We define an *atom over*  $\varphi$  (a  $\varphi$ -atom) to be a subset  $A \subseteq \Phi_\varphi$  satisfying the following requirements:

$R_{sat}$ : The conjunction of all state formulas in  $A$  is satisfiable.

$R_{\neg}$ : For every  $p \in \Phi_\varphi$ ,  $p \in A$  iff  $\neg p \notin A$ .

$R_\alpha$ : For every  $\alpha$ -formula  $p \in \Phi_\varphi$ ,  $p \in A$  iff  $\kappa(p) \subseteq A$ .

$R_\beta$ : For every  $\beta$ -formula  $p \in \Phi_\varphi$ ,  $p \in A$  iff either  $\kappa_1(p) \in A$  or  $\kappa_2(p) \subseteq A$  (or both).

For example, for the  $\alpha$ -formula  $\square p$ , requirement  $R_\alpha$  implies that  $\square p \in A$  iff both  $p \in A$  and  $\bigcirc \square p \in A$ . Note that a  $\varphi$ -atom must contain  $p$  or  $\neg p$ , for every  $p$  a subformula of  $\varphi$ .

An atom is supposed to represent a candidate set of temporal formulas that may possibly hold together at some position in a model. The inclusion of a formula  $p \in A$  means that  $p$  holds at that position. Its absence  $p \notin A$  means that  $p$  does not hold there. Indeed, requirement  $R_{\neg}$  stipulates that if  $p$  is absent from  $A$  then its negation  $\neg p$  is included in  $A$ .

**Example** Consider the formula

$$\varphi_1: \quad \square p \wedge \Diamond \neg p.$$

The set

$$A: \{ \square p \wedge \diamond \neg p, \quad \square p, \quad \diamond \neg p, \quad \circ \square p, \quad \circ \diamond \neg p, \quad p \}$$

is an atom of  $\varphi_1$ , while

$$B: \{ \square p \wedge \diamond \neg p, \quad \square p, \quad \diamond \neg p, \quad \circ \square p, \quad \neg \circ \diamond \neg p, \quad \neg p \}$$

is not an atom of  $\varphi_1$ . The set  $B$  is not an atom because, by requirement  $R_\alpha$ ,  $\square p \in B$  implies that  $p$  also belongs to  $B$ , which is not the case. ■

The tables of Fig. 5.1 do not include entries for formulas belonging to  $\Phi_\varphi^-$ , i.e., formulas whose principal operator is a negation. However, some requirements of the  $\alpha$ - and  $\beta$ -types are implied by requirements  $R_\alpha$  and  $R_\beta$  associated with their positive form and requirement  $R_\neg$ .

For example, we show that if atom  $A$  contains the formula  $\neg \diamond p$ , it must also contain the formulas  $\neg p$  and  $\neg \circ \diamond p$ . Observe that  $A$  cannot contain either  $p$  or  $\circ \diamond p$ . This is because, if  $A$  did contain any of them, it would also contain, by requirement  $R_\beta$  for  $\diamond p$ , the formula  $\diamond p$ . This would violate requirement  $R_\neg$  because then we would have both  $\diamond p \in A$  and  $\neg \diamond p \in A$ . We conclude that  $p \notin A$  from which follows, by requirement  $R_\neg$ , that  $\neg p \in A$ . In a similar way  $\circ \diamond p \notin A$ , from which follows  $\neg \circ \diamond p \in A$ .

In Fig. 5.2, we present the  $\alpha$ - and  $\beta$ -tables for the formulas  $\neg \diamond p$ ,  $\neg \square p$ , and  $\neg \exists p$ . All of them can be derived from the basic tables of Fig. 5.1 and the atom requirements.

$\alpha$	$\kappa(\alpha)$	$\beta$	$\kappa_1(\beta)$	$\kappa_2(\beta)$
$\neg \diamond p$	$\neg p, \neg \circ \diamond p$	$\neg \square p$	$\neg p$	$\neg \circ \square p$
$\neg \exists p$	$\neg p, \neg \circ \exists p$	$\neg \exists p$	$\neg p$	$\neg \circ \exists p$

Fig. 5.2.  $\alpha$ - and  $\beta$ -tables for some negations.

In Problem 5.1, the reader is requested to construct  $\alpha$ - and  $\beta$ -tables for the negation of the remaining formulas.

### Mutually Satisfiable Formulas

A set of formulas  $S \subseteq \Phi_\varphi$  is called *mutually satisfiable* if there exists a model  $\sigma$  and a position  $j \geq 0$ , such that every formula  $p \in S$  holds at position  $j$ .

The intended meaning of an atom is that it represents a maximal mutually satisfiable set of formulas. For example, requirement  $R_\alpha$  for  $\square p$  is justified by observing that if  $\square p$  holds at some position  $j$  in a model then both  $p$  and  $\circ \square p$  must also hold at that position.

This intended meaning is made more precise by the following claim:

**Claim 5.1** (atoms represent necessary conditions)

Let  $S \subseteq \Phi_\varphi$  be a mutually satisfiable set of formulas. Then there exists a  $\varphi$ -atom  $A$  such that  $S \subseteq A$ .

**Justification** Let  $S$  be a mutually satisfiable set of formulas in  $\Phi_\varphi$ . That is, there exists a model  $\sigma$ , and a position  $j \geq 0$ , such that every  $p \in S$  holds at  $j$ . Define the formula set

$$A = \{p \in \Phi_\varphi \mid (\sigma, j) \models p\}.$$

That is,  $A$  contains all the closure formulas that hold at position  $j$  of  $\sigma$ . It is straightforward to check that  $A$  is an atom. This is because each of the requirements an atom satisfies is implied by the definition of the  $\models$  relation. For example,  $p$  can hold at position  $j$  iff  $\neg p$  does not hold there, which implies that  $p \in A$  iff  $\neg p \notin A$ . Since all formulas in  $S$  are closure formulas that hold at  $j$ , we conclude that  $S \subseteq A$ . ■

It is important to realize that inclusion in an atom is only a necessary condition for mutual satisfiability. It is by no means a sufficient condition. For example, the set  $\{\varphi, \bigcirc p, \bigcirc \neg p, p\}$  is an atom of the formula  $\varphi$ :  $\bigcirc p \vee \bigcirc \neg p$ , but it is not a mutually satisfiable set. There is no model  $\sigma$  such that both  $\bigcirc p$  and  $\bigcirc \neg p$  hold at some position of  $\sigma$ .

### Basic Formulas

A formula is called *basic* if it is either an atomic formula (e.g., a proposition) or has the form  $\bigcirc p$ ,  $\ominus p$ , or  $\oslash p$ . The closure  $\Phi_\varphi$  of a formula  $\varphi$  contains no more than  $|\varphi|$  (size of  $\varphi$ ) basic formulas. This is because each subformula of  $\varphi$  can contribute at most one basic formula to the closure.

For example, the basic formulas in the closure of  $\varphi_1$ :  $\Box p \wedge \Diamond \neg p$  are

$$p, \quad \bigcirc \Box p, \quad \text{and} \quad \bigcirc \Diamond \neg p.$$

Basic formulas are important because their presence or absence in an atom uniquely determines (through the requirements an atom must satisfy) the presence or absence of all other closure formulas in the same atom.

**Example** (basic formulas determine the atom)

Assume we have already determined which of the three basic formulas:  $p$ ,  $\bigcirc \Box p$ ,  $\bigcirc \Diamond \neg p$ , are present in an atom  $A$  of the formula  $\varphi_1$ :  $\Box p \wedge \Diamond \neg p$ . Then we proceed as follows:

- For each  $\psi \in \{p, \bigcirc \Box p, \bigcirc \Diamond \neg p\}$ , if  $\psi \notin A$ , we place  $\neg\psi$  in  $A$ .
- If both  $p$  and  $\bigcirc \Box p$  are in  $A$ , we also place  $\Box p$  in  $A$ . Otherwise, we place  $\neg \Box p$  in  $A$ .

- If either  $\neg p$  or  $\bigcirc \diamond \neg p$  are in  $A$ , we place  $\diamond \neg p$  in  $A$ . Otherwise, we place  $\neg \diamond \neg p$  in  $A$ .
- If both  $\square p$  and  $\diamond \neg p$  are in  $A$ , we place  $\varphi_1: \square p \wedge \diamond \neg p$  in  $A$ . Otherwise, we place  $\neg \varphi_1$  in  $A$ .

Consider, for example, the case in which the basic formulas  $\bigcirc \square p$  and  $\bigcirc \diamond \neg p$  belong to  $A$  while  $p$  does not. This leads to the following  $\varphi_1$ -atom:

$$A: \{ \neg p, \bigcirc \square p, \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \}. \blacksquare$$

The fact that the interpretation of the basic formulas in an atom uniquely determines the atom suggests a systematic way to construct all atoms of a formula.

#### Algorithm ATOM — atom construction

- Let  $p_1, \dots, p_b \in \Phi_\varphi^+$  be all the basic formulas in the closure of formula  $\varphi$ .
- Construct all  $2^b$  combinations of the form  
 $q_1, \dots, q_b$ ,  
where  $q_i$  is either  $p_i$  or  $\neg p_i$ , for  $i = 1, \dots, b$ .
- Complete each combination into a full atom, using the atom requirements.

**Example** Consider again the formula  $\varphi_1: \square p \wedge \diamond \neg p$ , whose basic formulas are

$$p, \bigcirc \square p, \bigcirc \diamond \neg p.$$

Following is the list of all atoms of  $\varphi_1$ , constructed by Algorithm ATOM:

$$\begin{aligned} A_0: & \{ \neg p, \neg \bigcirc \square p, \neg \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \} \\ A_1: & \{ p, \neg \bigcirc \square p, \neg \bigcirc \diamond \neg p, \neg \square p, \neg \diamond \neg p, \neg \varphi_1 \} \\ A_2: & \{ \neg p, \neg \bigcirc \square p, \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \} \\ A_3: & \{ p, \neg \bigcirc \square p, \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \} \\ A_4: & \{ \neg p, \bigcirc \square p, \neg \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \} \\ A_5: & \{ p, \bigcirc \square p, \neg \bigcirc \diamond \neg p, \square p, \neg \diamond \neg p, \neg \varphi_1 \} \\ A_6: & \{ \neg p, \bigcirc \square p, \bigcirc \diamond \neg p, \neg \square p, \diamond \neg p, \neg \varphi_1 \} \\ A_7: & \{ p, \bigcirc \square p, \bigcirc \diamond \neg p, \square p, \diamond \neg p, \varphi_1 \}. \end{aligned}$$

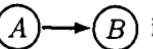
Note that atoms  $A_0$ – $A_6$  in the list all contain  $\neg \varphi_1$  while only  $A_7$  contains  $\varphi_1$  itself.  $\blacksquare$

Observe that if the closure of  $\varphi$  contains  $b$  basic formulas, there are precisely  $2^b$   $\varphi$ -atoms. Since  $b \leq |\varphi|$ , we conclude that the number of atoms for a formula  $\varphi$  is bounded by  $2^{|\varphi|}$ .

## The Tableau

Given a formula  $\varphi$ , we construct a directed graph  $T_\varphi$ , called the *tableau of  $\varphi$* , by the following algorithm.

### Algorithm TABLEAU — tableau construction

- The nodes of  $T_\varphi$  are the atoms of  $\varphi$ .
- Atom  $A$  is connected to atom  $B$  by a directed edge  if the following three *connection requirements* are satisfied:

$R_{\bigcirc}$ : For every  $\bigcirc p \in \Phi_\varphi$ ,  $\bigcirc p \in A$  iff  $p \in B$

$R_{\ominus}$ : For every  $\ominus p \in \Phi_\varphi$ ,  $p \in A$  iff  $\ominus p \in B$

$R_{\oslash}$ : For every  $\oslash p \in \Phi_\varphi$ ,  $p \in A$  iff  $\oslash p \in B$ .

If  $A$  is connected to  $B$ , we refer to  $B$  as a *successor* of  $A$ , and refer to  $A$  as a *predecessor* of  $B$ . These requirements identify local necessary conditions for the situation that all the formulas contained in  $A$  hold at some position  $j$ , while all the formulas of  $B$  hold in the immediately next position  $j + 1$ . Indeed, if  $\bigcirc p$  holds at position  $j$  then  $p$  should hold at position  $j + 1$ , and if  $\ominus p$  or  $\oslash p$  hold at  $j + 1$  then  $p$  should hold at  $j$ .

**Examples** In Fig. 5.3 we present the tableau for the formula

$$\varphi_1: \quad \square p \wedge \diamond \neg p.$$

This representation uses encapsulation conventions (see page 277) for describing connections between atoms. According to these conventions, each of  $\{A_2, A_3\}$  is connected to  $A_0, A_2, A_3, A_4$ , and  $A_6$ .

Consider atoms  $A_2$  and  $A_3$ . Since both atoms contain the formula  $\bigcirc \diamond \neg p$ , each of their successors should contain the formula  $\diamond \neg p$ . No successor of  $A_2$  or  $A_3$  can contain the formula  $\square p$ . This is because any predecessor of an atom containing  $\square p$  must contain the formula  $\bigcirc \square p$  and cannot, therefore, contain  $\neg \bigcirc \square p$  which both  $A_2$  and  $A_3$  contain. It follows that any successor of  $A_2$  and  $A_3$  must contain both  $\diamond \neg p$  and  $\neg \square p$ . This explains why  $A_2$  and  $A_3$  are connected to all of  $\{A_0, A_2, A_3, A_4, A_6\}$  which are all the atoms containing  $\{\diamond \neg p, \neg \square p\}$ . ■

An atom  $A$  is called *initial* if it does not contain a formula of the form  $\ominus p$  or a formula of the form  $\neg \oslash p$ . An initial atom represents a candidate set of

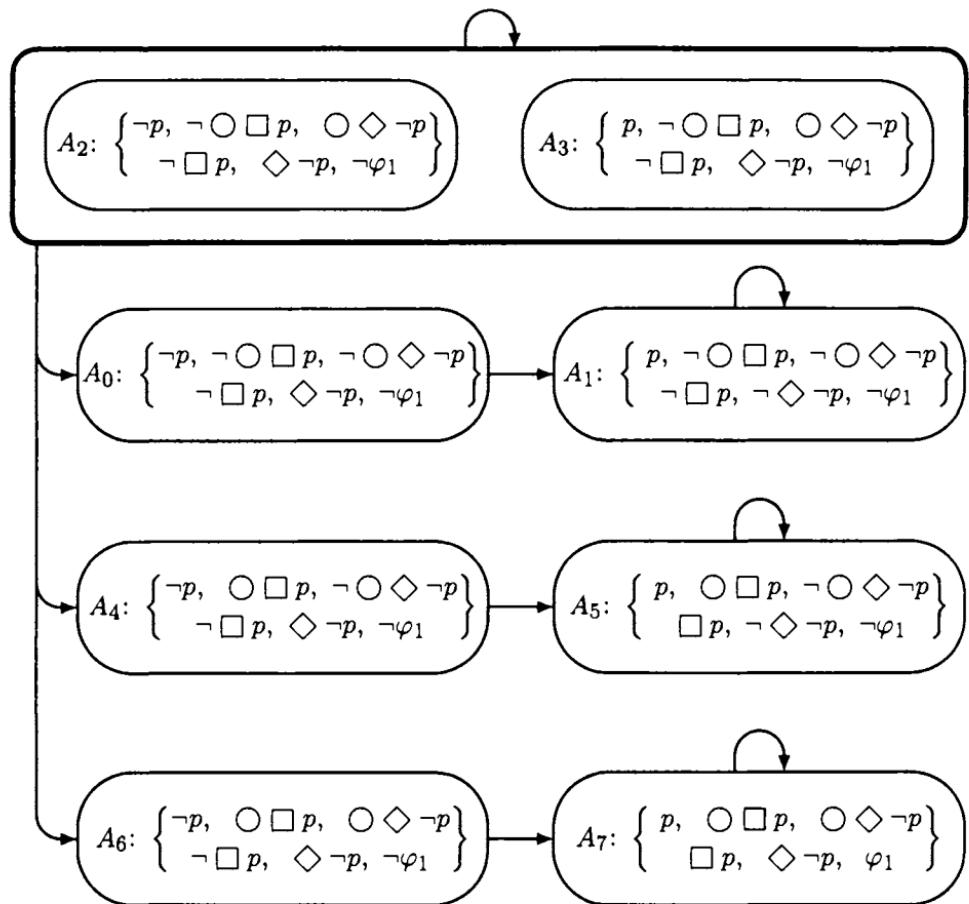


Fig. 5.3. Tableau  $T_{\varphi_1}$  for formula  $\varphi_1: \Box p \wedge \Diamond \neg p$ .

closure formulas that may possibly hold at position 0 of a model. Atoms  $A_0$ – $A_7$  of the formula  $\varphi_1: \Box p \wedge \Diamond \neg p$  are initial.

Note that an initial atom can still contain a formula of the form  $\Diamond p$ , or even the set  $\{\Diamond p, \Diamond \neg p\}$ . An initial atom that contains the formula  $\varphi$  is called an *initial  $\varphi$ -atom*.

## Models vs. Paths

Our interest in the tableau stems from the fact that it is a finite structure which embeds all possible models. The following claim establishes a connection between models and paths in the tableau.

## Paths Induced by Models

For a model  $\sigma$ , the infinite atom path  $\pi_\sigma: A_0, A_1, \dots$  in  $T_\varphi$  is said to be *induced* by  $\sigma$  if, for every position  $j \geq 0$  and every closure formula  $p \in \Phi_\varphi$ ,

$$(\sigma, j) \models p \text{ iff } p \in A_j.$$

### Claim 5.2 (models induce paths)

Consider a formula  $\varphi$  and its tableau  $T_\varphi$ . For every model  $\sigma: s_0, s_1, \dots$ , there exists an infinite atom path  $\pi_\sigma: A_0, A_1, \dots$  in  $T_\varphi$  induced by  $\sigma$ .

Furthermore,  $A_0$  is an initial atom, and if  $\sigma \models \varphi$  then  $\varphi \in A_0$ .

**Justification** Let  $\sigma: s_0, s_1, \dots$  be a model. For each  $j \geq 0$ , define  $A_j$  to be the subset of  $\Phi_\varphi$  that contains all formulas  $p \in \Phi_\varphi$  such that  $(\sigma, j) \models p$ . It can be established that  $A_j$  satisfies all the requirements expected of an atom. Furthermore, we can show that the conditions for connecting atom  $A$  to atom  $B$  are satisfied by  $A = A_j$  and  $B = A_{j+1}$ , for every  $j \geq 0$ .

Thus,  $\pi_\sigma: A_0, A_1, \dots$  is an infinite path in the tableau  $T_\varphi$ , induced by the model  $\sigma$ .

Atom  $A_0$  is always initial, since position 0 can never satisfy any formula of the form  $\Theta p$ .

If, in addition,  $\sigma \models \varphi$  then  $(\sigma, 0) \models \varphi$  and, by definition of  $A_j$ ,  $\varphi \in A_0$ . ■

The converse of Claim 5.2 is not necessarily true. Not every infinite path in the tableau is induced by some model  $\sigma$ . For example, the periodic path  $A_7^\omega: A_7, A_7, \dots$  in the tableau of Fig. 5.3 cannot be induced by a model. To see this, assume that there was a model  $\sigma: s_0, s_1, \dots$  that induces  $A_7^\omega$ . By the definition of paths induced by models, every formula  $q \in A_7$  should hold at all positions of  $\sigma$ . In particular,  $\Diamond \neg p$  holds at position 0 and  $p$  holds at all positions  $j \geq 0$ . This contradicts the semantics of  $\Diamond \neg p$ .

To get a correspondence in the other direction, we have to restrict our attention to paths that satisfy additional conditions.

## Promising Formulas

A formula  $\psi \in \Phi_\varphi$  is said to *promise* the formula  $r$  if  $\psi$  has one of the following forms:

$$\Diamond r, \quad p \vee r, \quad \neg \Box \neg r, \quad \neg((\neg r) \wedge p),$$

or if  $r$  is the negation  $\neg q$  and  $\psi$  has one of the forms:

$$\neg \Box q, \quad \neg(q \wedge p),$$

for an arbitrary formula  $p$ . Observe that each of these six formulas implies  $\Diamond r$ , which can be viewed as a promise that  $r$  will eventually hold. We refer to such

a formula  $\psi$  as a *promising* formula. For example, the closure  $\Phi_{\varphi_1}$  contains the promising formulas  $\Diamond \neg p$  and  $\neg \Box p$ , both promising  $r: \neg p$ .

Let  $\sigma$  be a model and  $\psi$  a formula promising (the subformula)  $r$ . From the semantics of the various forms of promising formulas, we can infer the following obvious fact:

For every  $j \geq 0$ , if  $(\sigma, j) \models \psi$ , then  $(\sigma, k) \models r$  for some  $k \geq j$ .

That is, all promises for  $r$  are eventually fulfilled. The following claim presents a consequence of this observation.

**Claim 5.3** (promise fulfillment by models)

Let  $\sigma$  be a model and  $\psi$ , a formula promising  $r$ . Then,  $\sigma$  contains infinitely many positions  $j \geq 0$  such that

$$(\sigma, j) \models \neg \psi \quad \text{or} \quad (\sigma, j) \models r.$$

**Justification** We consider two cases. If  $\sigma$  contains infinitely many  $\psi$ -positions then, since each of them must be followed by an  $r$ -position,  $\sigma$  contains infinitely many  $r$ -positions. If  $\sigma$  contains only finitely many  $\psi$ -positions, then it certainly contains infinitely many  $(\neg \psi)$ -positions. Thus, in any case,  $\sigma$  contains infinitely many  $(\neg \psi \vee r)$ -positions. ■

## Fulfilling Paths

We wish to restrict our attention to paths in the tableau that reproduce this property. For that purpose, we introduce several definitions.

We say that atom  $A$  *fulfills* a formula  $\psi$  that promises  $r$  if  $\neg \psi \in A$  or  $r \in A$ . A path  $\pi: A_0, A_1, \dots$  in the tableau  $T_\varphi$  is called *fulfilling* if  $A_0$  is an initial atom and, for every promising formula  $\psi \in \Phi_\varphi$ ,  $\pi$  contains infinitely many atoms  $A_j$  that fulfill  $\psi$ .

**Example** Consider again the tableau of Fig. 5.3. We observe that the path  $A_7^\omega$  is not fulfilling, since  $A_7$  contains the formula  $\Diamond \neg p$  which promises  $\neg p$  but does not fulfill this promise. In a similar way, neither is  $A_1^\omega$ , which promises  $\neg p$  (through  $\neg \Box p$ ), nor  $A_3^\omega$ , which promises  $\neg p$ , fulfilling. On the other hand, both  $A_2^\omega$  and  $(A_2, A_3)^\omega$  are fulfilling. They promise  $\neg p$  but infinitely often visit  $A_2$  which fulfills  $\neg p$ .

Path  $A_4, A_5^\omega$  is also fulfilling since  $A_5$  contains the negation of the two promising formulas:  $\Diamond \neg p$  and  $\neg \Box p$ . ■

**Claim 5.4** (models induce fulfilling paths)

If  $\pi_\sigma: A_0, A_1, \dots$  is a path induced by a model  $\sigma$ , then  $\pi_\sigma$  is fulfilling.

**Justification** Let  $\psi \in \Phi_\varphi$  be a formula that promises  $r$ . By the property of models,  $\sigma$  contains infinitely many positions  $j$  such that  $(\sigma, j) \models \neg\psi$  or  $(\sigma, j) \models r$ . By the correspondence between a model and the path it induces, for each of these positions  $j$ ,  $\neg\psi \in A_j$  or  $r \in A_j$ . ■

This shows that fulfillment is a necessary condition a path must satisfy in order to be induced by a model. It is also a sufficient condition.

**Claim 5.5** (fulfilling paths induce models)

If  $\pi: A_0, A_1, \dots$  is a fulfilling path in  $T_\varphi$ , there exists a model  $\sigma$  inducing  $\pi$ , i.e.,  $\pi = \pi_\sigma$  and, for every  $\psi \in \Phi_\varphi$  and every  $j \geq 0$ ,

$(\sigma, j) \models \psi$  iff  $\psi \in A_j$ .

**Justification** We define the model  $\sigma$ :  $s_0, s_1, \dots$  corresponding to the fulfilling path  $\pi$  by ensuring that  $s_j$  and  $A_j$  agree on the interpretation of the propositions in  $\Phi_\varphi$ . Thus, for each  $j \geq 0$  and each proposition  $p \in \Phi_\varphi$ , we let  $s_j[p] = \top$  if  $p \in A_j$ , and  $s_j[p] = \perp$  otherwise (in which case  $\neg p \in A_j$ ).

To show that  $(\sigma, j) \models \psi$  iff  $\psi \in A_j$ , we proceed by induction on the structure of  $\psi \in \Phi_\varphi$ . The case that  $\psi$  is a proposition is trivial, since  $\sigma$  was defined to ensure that  $\sigma$  and  $\pi$  agree on the interpretation of propositions. It is straightforward to see that if  $\sigma$  and  $\pi$  agree on propositions, they also agree on any boolean combination of propositions. The induction proceeds by considering application of the different temporal operators. We will illustrate the proof on two cases.

- $$\bullet \quad \psi = \bigcirc p.$$

Assume that we have already established that  $\sigma$  and  $\pi$  agree on the interpretation of  $p$ . Then, we argue as follows:

$$(\sigma, j) \models \bigcirc p \quad \xleftarrow{\text{def of } \bigcirc} \quad (\sigma, j+1) \models p \quad \xleftarrow{\text{Induction hyp.}} \quad p \in A_{j+1}$$

Conditions for connecting  $A_j$  to  $A_{j+1}$

$$\xleftarrow{\quad\quad\quad} \quad \bigcirc p \in A_j.$$

- $$\psi = \diamond r.$$

Assume that  $\sigma$  and  $\pi$  have been shown to agree on  $r$ . We break the argument into two parts, showing first that  $\Diamond r \in A_j$  implies  $(\sigma, j) \models \Diamond r$  and then that  $(\sigma, j) \models \Diamond r$  implies  $\Diamond r \in A_j$ .

Assume that  $\Diamond r \in A_j$ . Since  $\pi$  is fulfilling, it contains infinitely many positions  $k$  beyond  $j$  such that  $A_k$  fulfills  $\Diamond r$ . Let  $k$  be the smallest index  $k \geq j$  such that  $A_k$  fulfills  $\Diamond r$ . If  $k = j$  then, since  $\Diamond r \in A_j$ ,  $A_k$  can fulfill  $\Diamond r$  only by having  $r \in A_k$ . If  $k > j$  then  $A_{k-1}$  does not fulfill  $\Diamond r$  and must, therefore, contain both  $\Diamond r$  and  $\neg r$ . By requirement  $R_\beta$  for  $\Diamond r$ ,  $\Box \Diamond r \in A_{k-1}$  and hence  $\Diamond r \in A_k$ . The only way  $A_k$  can fulfill  $\Diamond r$  is to have  $r \in A_k$ . Consequently, there always

exists a  $k \geq j$  such that  $r \in A_k$ . By the induction hypothesis,  $(\sigma, k) \models r$ , which, by the definition of  $\Diamond r$ , implies  $(\sigma, j) \models \Diamond r$ .

In the other direction, assume that  $(\sigma, j) \models \Diamond r$  but  $\Diamond r \notin A_j$ . In that case,  $\neg \Diamond r \in A_j$  which implies, by the requirements from atoms and their successors, that  $\{\neg \Diamond r, \neg r\} \subseteq A_k$  for all  $k \geq j$ . By the induction hypothesis this implies that  $(\sigma, k) \models \neg r$  for all  $k \geq j$ , which contradicts  $(\sigma, j) \models \Diamond r$ . ■

**Example** For an arbitrary  $m > 0$ , consider the path

$$\pi: (A_2, A_3)^m, A_4, A_5^\omega$$

in the tableau of Fig. 5.3. This path visits  $A_2$  and  $A_3$  precisely  $m$  times, it then visits  $A_4$  once, and proceeds to repeat  $A_5$  indefinitely. This path is fulfilling since atom  $A_5$ , which is visited infinitely many times, contains the negation of the only two promising formulas in  $\Phi_{\varphi_1}$ :  $\neg \Box p$  and  $\Diamond \neg p$ . Consequently, the model

$$\sigma: (\langle p: F \rangle, \langle p: T \rangle)^m, \langle p: F \rangle, \langle p: T \rangle^\omega$$

induces the path  $\pi$ . ■

In **Problem 5.2**, the reader is requested to show that no requirement of fulfillment is necessary for past formulas such as  $\Diamond r$  or  $p \mathcal{S} r$ .

Claims 5.2, 5.4, and 5.5 can be combined to yield the following result:

**Proposition 5.6** (satisfiability and fulfilling paths)

Formula  $\varphi$  is satisfiable iff the tableau  $T_\varphi$  contains a fulfilling path

$$\pi: A_0, A_1, \dots$$

such that  $A_0$  is an initial  $\varphi$ -atom.

**Justification** Assume that  $\pi$  is a fulfilling path such that  $\varphi \in A_0$ . By Claim 5.5 there exists a model  $\sigma$  such that  $(\sigma, j) \models p$  iff  $p \in A_j$ , for every  $j \geq 0$  and  $p \in \Phi_\varphi$ . In particular,  $(\sigma, 0) \models \varphi$  which shows that  $\sigma$  satisfies  $\varphi$ .

In the other direction, let  $\sigma$  be a model satisfying  $\varphi$ . By Claims 5.2 and 5.4,  $\sigma$  induces a fulfilling path  $\pi_\sigma: A_0, A_1, \dots$ . Since  $(\sigma, 0) \models \varphi$ , the correspondence between  $\sigma$  and  $\pi_\sigma$  implies that  $\varphi \in A_0$ . ■

**Example** Let us check whether the formula

$$\varphi_1: \Box p \wedge \Diamond \neg p$$

is satisfiable. According to Proposition 5.6, it is sufficient to check whether the tableau for  $\varphi_1$ , presented in Fig. 5.3, contains a fulfilling path  $\pi: B_0, B_1, \dots$  such that  $\varphi_1 \in B_0$ . The only atom containing  $\varphi_1$  in this tableau is  $A_7$ , and the only infinite path starting at  $A_7$  is  $A_7^\omega$ . As it was previously observed that  $A_7^\omega$  is not fulfilling, we conclude that  $\varphi_1$  is unsatisfiable.

As another example, let us check for the satisfiability of

$$\psi_1 = \neg\varphi_1: \neg(\Box p \wedge \Diamond \neg p),$$

which can also be written as  $\neg\Box p \vee \neg\Diamond \neg p$ . It is not difficult to see that the tableaux for  $\varphi$  and  $\neg\varphi$  are always identical. Therefore, we can use again the tableau of Fig. 5.3, but, this time, searching for a fulfilling path  $\pi: B_0, B_1, \dots$  such that  $\psi_1 = \neg\varphi_1 \in B_0$ . Now we have better luck, since  $A_5^\omega$  is a fulfilling path such that  $\neg\varphi_1$  belongs to its first atom. Indeed the corresponding model

$$\langle p: T \rangle^\omega$$

satisfies  $\psi_1: \neg(\Box p \wedge \Diamond \neg p)$ . ■

## Strongly Connected Subgraphs

Proposition 5.6 reduced the search for a satisfying model to a search of fulfilling paths in the tableau  $T_\varphi$ . Unfortunately, this does not yet yield an effective way for testing satisfiability of a formula  $\varphi$  since, in general, a tableau may contain infinitely many different paths.

For example, the tableau of Fig. 5.3 contains the infinite family of fulfilling paths  $\{(A_2, A_3)^m, A_4, A_5^\omega \mid m \geq 0\}$ . This indicates that a further reduction is necessary.

We recall from Section 3.6 that a subgraph  $S \subseteq T_\varphi$  is called a *strongly connected subgraph* (scs) if for every two distinct atoms  $A, B \in S$ , there exists a path from  $A$  to  $B$  that passes through atoms of  $S$ . For example,  $\{A_2\}$ ,  $\{A_3\}$ , and  $\{A_2, A_3\}$  are all scs's of the tableau of Fig. 5.3.

An scs  $S$  is called *transient* if it consists of a single atom that is not connected to itself. For example,  $\{A_4\}$  of Fig. 5.3 is a transient scs, while  $\{A_5\}$  is not.

An scs  $S$  is called *fulfilling* if it is non-transient and every promising formula  $\psi \in \Phi_\varphi$  is fulfilled by some atom  $A \in S$ . For example, in Fig. 5.3, the scs's  $\{A_2\}$ ,  $\{A_2, A_3\}$ , and  $\{A_5\}$  are fulfilling while  $\{A_1\}$ ,  $\{A_3\}$ ,  $\{A_4\}$ , and  $\{A_7\}$  are strongly connected but not fulfilling. The scs  $\{A_4\}$  is not fulfilling because it is transient, while scs  $\{A_7\}$  is not fulfilling since it contains the promising formula  $\Diamond \neg p$  but does not contain  $\neg p$ . Subgraphs  $\{A_1\}$  and  $\{A_3\}$  are not fulfilling since each contains the promising formula  $\neg \Box p$  but does not contain the promised  $\neg p$ .

The scs  $S$  is said to be  $\varphi$ -*reachable* if there exists a finite path  $B_0, B_1, \dots, B_k$  such that  $B_0$  is an initial  $\varphi$ -atom and  $B_k \in S$ . For example,  $\{A_7\}$  is the only  $\varphi_1$ -reachable scs in the tableau of Fig. 5.3, while the list of  $\neg\varphi_1$ -reachable scs's in the same tableau is given by

$$\{A_0\}, \{A_1\}, \{A_2\}, \{A_3\}, \{A_2, A_3\}, \{A_4\}, \{A_5\}, \{A_6\}, \{A_7\}.$$

The following claim shows that instead of searching for fulfilling paths it is

sufficient to search for fulfilling SCS's.

**Claim 5.7** (fulfilling paths and subgraphs)

The tableau  $T_\varphi$  contains a fulfilling path starting at an initial  $\varphi$ -atom iff  $T_\varphi$  contains a  $\varphi$ -reachable fulfilling SCS.

**Justification** Assume that  $T_\varphi$  contains a fulfilling path  $\pi: A_0, A_1, \dots$ , such that  $\varphi \in A_0$ . Let  $S$  be the set of atoms that appear infinitely many times in  $\pi$ . We show that  $S$  is a  $\varphi$ -reachable fulfilling SCS.

- $S$  is *strongly connected* — Since each atom that appears in  $\pi$  but does not belong to  $S$  appears at only finitely many positions in  $\pi$ , it is possible to identify a cutoff position  $f$ , such that  $A_i \in S$  for every  $i \geq f$ . That is, all atoms appearing beyond position  $f$  appear infinitely often in  $\pi$ . Let  $A, B \in S$  be two distinct atoms. Since each of them appears infinitely often in  $\pi$ , we can find positions  $j$  and  $k$ ,  $f < j < k$ , such that  $A_j = A$  and  $A_k = B$ . It is obvious that the path  $A = A_j, A_{j+1}, \dots, A_k = B$  passes only through  $S$ -atoms and connects  $A$  to  $B$ . In the case that  $S$  contains a single atom, it is strongly connected by definition.
- $S$  is *fulfilling* — Consider any promising formula  $\psi \in \Phi_\varphi$ . Since  $\pi$  is fulfilling, there exist infinitely many positions  $j$  such that  $A_j$  fulfills  $\psi$ . As there are only finitely many different atoms in  $T_\varphi$ , there must be an atom  $A$  fulfilling  $\psi$  that appears at infinitely many positions in  $\pi$ . Obviously  $A \in S$ . This shows that every promising formula in  $\Phi_\varphi$  is fulfilled by some atom in  $S$ .
- $S$  is  *$\varphi$ -reachable* — Since  $A_f \in S$ , the finite path  $A_0, A_1, \dots, A_f$  leads from an initial  $\varphi$ -atom to an atom in  $S$ .

In the other direction, assume that  $S$  is a  $\varphi$ -reachable fulfilling SCS. As  $S$  is  $\varphi$ -reachable, there exists a finite path  $B_0, \dots, B_j$ , such that  $B_0$  is an initial  $\varphi$ -atom, and  $B_j \in S$ . Strong connectedness and non-transience of  $S$  imply the existence of a cyclic traversing path

$$B_j, B_{j+1}, \dots, B_k = B_j$$

that passes only through atoms of  $S$  and visits each atom of  $S$  at least once. Then, the ultimately periodic infinite path

$$\pi: B_0, \dots, B_j, (B_{j+1}, \dots, B_k)^\omega$$

starts at an initial  $\varphi$ -atom and is fulfilling. ■

For a path  $\pi: A_0, A_1, \dots$ , we denote by  $Inf(\pi)$  the SCS consisting of the atoms that appear infinitely many times in  $\pi$ .

For example, the set  $S = \{A_2, A_3\}$  is a  $(\neg\varphi_1)$ -reachable fulfilling SCS in the tableau of Fig. 5.3. Indeed, it can be traversed by the infinite path

$$(A_2, A_3)^\omega$$

which starts at an initial ( $\neg\varphi_1$ )-atom.

In principle, Claim 5.7 provides a solution to the satisfiability checking problem. It is sufficient to construct the tableau  $T_\varphi$  and check whether any of its SCS's, of which there are only a finite number, is  $\varphi$ -reachable and fulfilling.

**Example** Consider the tableau of Fig. 5.3. The SCS's of this tableau are

$$\{A_0\}, \{A_1\}, \{A_2\}, \{A_3\}, \{A_4\}, \{A_5\}, \{A_6\}, \{A_7\}, \{A_2, A_3\}.$$

Of these, only  $\{A_2\}$ ,  $\{A_3\}$ ,  $\{A_2, A_3\}$ , and  $\{A_5\}$  are fulfilling. If we are checking for satisfiability of  $\varphi_1$ :  $\Box p \wedge \Diamond \neg p$ , we conclude that this formula is unsatisfiable since none of the fulfilling SCS's is reachable from an initial  $\varphi_1$ -atom.

On the other hand, if we are checking for satisfiability of  $\neg\varphi_1$ :  $\neg(\Box p \wedge \Diamond \neg p)$ , we conclude that this formula is satisfiable since the fulfilling SCS  $A_2$  is  $\neg\varphi_1$ -reachable. ■

Unfortunately, while the number of different SCS's in  $T_\varphi$  is finite, it may be exponential in the size of  $T_\varphi$ , denoted by  $|T_\varphi|$ . Can we do better?

## Maximal Strongly Connected Subgraphs

An SCS  $S$  is said to be a *maximal strongly connected subgraph* (MSCS) if it is not contained in any larger SCS. For example,  $\{A_2, A_3\}$  is an MSCS, while  $\{A_2\}$  and  $\{A_3\}$  are not.

We can restrict our search for  $\varphi$ -reachable fulfilling SCS's in  $T_\varphi$  to the maximal ones, whose number is bounded by  $|T_\varphi|$ .

As in Section 3.6, we will use Algorithm DECOMPOSE (pages 299, 315), which inputs a graph  $S$  and outputs its sorted decomposition into MSCS's  $U_1, \dots, U_k$ .

Let  $S$  be a  $\varphi$ -reachable fulfilling SCS. If  $S$  is not a maximal SCS (an MSCS), it is contained in a unique MSCS  $S' \supseteq S$ . It is not difficult to see that  $S'$  is also  $\varphi$ -reachable and fulfilling. In fact, the same path  $B_0, \dots, B_k \in S$ ,  $\varphi \in B_0$ , that shows  $\varphi$ -reachability for  $S$  also demonstrates  $\varphi$ -reachability for  $S'$ . In a similar way, every atom  $A \in S$  that fulfills a promising formula  $\psi$ , also belongs to  $S'$  as  $S \subseteq S'$ .

This leads to the following algorithm:

### Algorithm SAT — satisfiability of a formula

To check whether a temporal formula  $\varphi$  is satisfiable,

- Construct the tableau  $T_\varphi$ .
- Construct the graph  $T_\varphi^-$  obtained by removing all atoms that are not reachable from an initial  $\varphi$ -atom.

- Using Algorithm DECOMPOSE, decompose  $T_\varphi^-$  into MSCS's  $U_1, \dots, U_k$ .
- For each  $i = 1, \dots, k$ , check whether  $U_i$  is fulfilling. If some  $U_i$  is fulfilling, report *success* —  $\varphi$  is satisfiable. The path consisting of an initial segment  $B_0, \dots, B_k \in U_i$ ,  $\varphi \in B_0$ , followed by an endless repetition of a cyclic path traversing  $U_i$ , defines a model  $\sigma$  satisfying  $\varphi$ .
- If no  $U_i$  is fulfilling, report *failure* —  $\varphi$  is unsatisfiable.

The correctness of the algorithm is established by the following proposition.

**Proposition 5.8** (satisfiability and MSCS's)

A formula  $\varphi$  is satisfiable iff the tableau  $T_\varphi$  contains a  $\varphi$ -reachable fulfilling MSCS.

**Justification** Assume that  $\varphi$  is satisfiable. By Proposition 5.6 and Claim 5.7,  $T_\varphi$  contains a  $\varphi$ -reachable fulfilling SCS  $S$ . By the discussion preceding the algorithm, there exists a  $\varphi$ -reachable fulfilling MSCS  $S'$  containing  $S$ , as required by the proposition.

In the other direction, assume that  $T_\varphi$  contains a  $\varphi$ -reachable fulfilling MSCS  $S$ . Since  $S$  is a special case of an SCS, Claim 5.7 and Proposition 5.6 imply that  $\varphi$  is satisfiable. ■

**Example** Let us check whether  $\varphi_1: \square p \wedge \Diamond \neg p$  is satisfiable. The tableau  $T_{\varphi_1}$  is presented in Fig. 5.3. Removing atoms that are not  $\varphi_1$ -reachable, we obtain  $T_{\varphi_1}^-$  which consists of  $A_7$  alone. Decomposing into MSCS's yields  $\{A_7\}$  as the only MSCS. Since  $A_7$  is not fulfilling, we conclude that  $\varphi_1$  is unsatisfiable.

As a second example, let us check whether  $\psi_1 = \neg \varphi_1$  is satisfiable. As previously observed, the tableau for  $\psi_1$  is identical to  $T_{\varphi_1}$  of Fig. 5.3. However,  $T_{\psi_1}^-$  consists of all atoms. Decomposing  $T_{\psi_1}^-$ , we obtain the following list of MSCS's

$$\{A_0\}, \{A_1\}, \{A_2, A_3\}, \{A_4\}, \{A_5\}, \{A_6\}, \{A_7\}.$$

Of these,  $\{A_2, A_3\}$  and  $\{A_5\}$  are fulfilling. Therefore  $\psi_1$  is satisfiable. Choosing the smaller  $\{A_5\}$ , we obtain the model

$$\sigma: \langle p: T \rangle^\omega$$

which obviously satisfies  $\psi_1: \neg(\square p \wedge \Diamond \neg p)$ . ■

Algorithm SAT can also be used to check validity. To check whether  $\varphi$  is valid, we apply Algorithm SAT to  $\neg \varphi$ . If the algorithm reports success,  $\varphi$  is not valid and the produced model  $\sigma$  is a counterexample. If the algorithm reports failure,  $\neg \varphi$  is unsatisfiable and, therefore,  $\varphi$  is valid.

## Complexity

Let us assess the complexity of Algorithm SAT. Let  $n = |\varphi|$  denote the size of the examined formula and  $a$  denote the number of atoms in  $T_\varphi$ . As previously observed,  $a \leq 2^n$ . The construction of the tableau has to consider  $a$  atoms and at most  $a^2$  connections (edges). The amount of work for each atom and connection is bounded by  $O(n)$ . Thus, the tableau can be constructed by  $O(na^2)$  steps. The removal of atoms that are not  $\varphi$ -reachable and the application of Algorithm DECOMPOSE together take a number of steps that can be bounded by  $O(a^2)$ . Checking the MSCS's for fulfillment takes no more than  $O(na)$  steps. We conclude that an upper bound for the number of steps taken by Algorithm SAT is  $O(na^2 + a^2 + na)$  which is asymptotically dominated by  $O(n2^n)$ .

In **Problem 5.3**, the reader is requested to show that, for some formulas of size  $n$ , it is impossible to construct a tableau with fewer than  $\Omega(2^n)$  atoms.

### Example (non-propositional tableau)

All the previous examples of formulas and their tableaux were propositional. We consider now the non-propositional formula  $\text{at\_}\ell_2 \Rightarrow \Diamond \text{at\_}\ell_3$ .

In terms of our basic operators set, this formula can be written as:

$$\varphi_2: \quad \square \left( \underbrace{\neg \text{at\_}\ell_2 \vee \Diamond \text{at\_}\ell_3}_{p_2} \right).$$

The closure of this formula is given by  $\Phi_{\varphi_2} = \Phi_{\varphi_2}^+ \cup \Phi_{\varphi_2}^-$ , where

$$\Phi_{\varphi_2}^+: \{ \square p_2, \bigcirc \square p_2, p_2, \neg \text{at\_}\ell_2, \Diamond \text{at\_}\ell_3, \bigcirc \Diamond \text{at\_}\ell_3, \text{at\_}\ell_3 \}.$$

Since we intend to retain only the  $\varphi_2$ -reachable atoms in the tableau, we observe that all such atoms must contain the formulas  $\square p_2$ ,  $p_2$ , and  $\bigcirc \square p_2$ . This is because  $\varphi_2$  has the form  $\square p_2$  and therefore any atom containing  $\varphi_2$  must also contain (by  $R_\square$ )  $p_2$  and  $\bigcirc \square p_2$ . Thus, any atom connected to an atom containing  $\varphi_2$  must contain (by  $R_\bigcirc$ )  $\square p_2$ , and therefore also  $p_2$  and  $\bigcirc \square p_2$ . It follows that any atom reachable from a  $\varphi_2$ -atom must contain  $\square p_2$ ,  $p_2$ , and  $\bigcirc \square p_2$ . Therefore, we enumerate only such atoms. They are given by

- $A_0: \{ \square p_2, \bigcirc \square p_2, p_2, \neg \text{at\_}\ell_2, \neg \text{at\_}\ell_3, \neg \bigcirc \Diamond \text{at\_}\ell_3, \neg \Diamond \text{at\_}\ell_3 \}$
- $A_1: \{ \square p_2, \bigcirc \square p_2, p_2, \neg \text{at\_}\ell_2, \neg \text{at\_}\ell_3, \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$
- $A_2: \{ \square p_2, \bigcirc \square p_2, p_2, \neg \text{at\_}\ell_2, \text{at\_}\ell_3, \neg \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$
- $A_3: \{ \square p_2, \bigcirc \square p_2, p_2, \neg \text{at\_}\ell_2, \text{at\_}\ell_3, \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$
- $A_4: \{ \square p_2, \bigcirc \square p_2, p_2, \text{at\_}\ell_2, \neg \text{at\_}\ell_3, \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$
- $A_5: \{ \square p_2, \bigcirc \square p_2, p_2, \text{at\_}\ell_2, \text{at\_}\ell_3, \neg \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$
- $A_6: \{ \square p_2, \bigcirc \square p_2, p_2, \text{at\_}\ell_2, \text{at\_}\ell_3, \bigcirc \Diamond \text{at\_}\ell_3, \Diamond \text{at\_}\ell_3 \}$

Note that the atom containing  $\{at\_l_2, \neg at\_l_3, \neg \bigcirc \diamondsuit at\_l_3\}$  is not considered since  $p_2: \neg at\_l_2 \vee \diamondsuit at\_l_3$  and  $at\_l_2$  imply  $\diamondsuit at\_l_3$  while  $\neg at\_l_3$  and the formula  $\neg \bigcirc \diamondsuit at\_l_3$  imply  $\neg \diamondsuit at\_l_3$ .

In Fig. 5.4, we present the tableau for formula  $\varphi_2$ . This representation also uses encapsulation conventions for describing connections between atoms. According to these conventions, each of  $\{A_1, A_3, A_4, A_6\}$  is connected to each of  $\{A_1, A_3, A_4, A_6, A_2, A_5\}$ . Each of  $\{A_2, A_5\}$  is connected to  $A_0$ , which is only connected to itself.

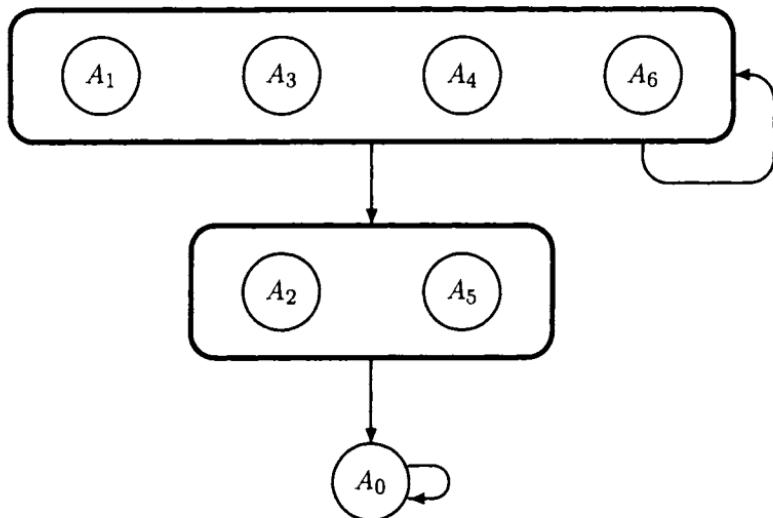


Fig. 5.4. Tableau for formula  $\varphi_2: \square(\neg at\_l_2 \vee \diamondsuit at\_l_3)$ .

Decomposing the tableau, we obtain the following MSCS's:

$$\{A_1, A_3, A_4, A_6\}, \{A_2\}, \{A_5\}, \{A_0\}.$$

The only promising formula in these atoms is  $\diamondsuit at\_l_3$  and it is fulfilled by atoms  $A_0, A_2, A_3, A_5$ , and  $A_6$ . Atom  $A_0$  fulfills  $\diamondsuit at\_l_3$  since  $\diamondsuit at\_l_3 \notin A_0$ . Atoms  $A_2, A_3, A_5$ , and  $A_6$  fulfill  $\diamondsuit at\_l_3$  by containing  $at\_l_3$ . Since  $\{A_2\}$  and  $\{A_5\}$  are transient, we are left with  $\{A_1, A_3, A_4, A_6\}$  and  $\{A_0\}$  as the fulfilling MSCS's of this tableau. Note that the closure for  $\varphi_2$  also contains the promising formula  $\neg \varphi_2$ . However, since  $\neg \varphi_2$  does not appear in any of the considered atoms, it is trivially fulfilled.

The formula  $\varphi_2$  is therefore satisfiable, e.g., by the model

$$(at\_l_2 : F, at\_l_3 : F)^\omega,$$

which is obtained by infinitely traversing  $A_0$ . ■

## Pruning the Tableau

After constructing the tableau  $T_\varphi$  for a formula  $\varphi$ , it is possible to remove atoms that obviously cannot contribute to the satisfaction of  $\varphi$  from the tableau. These are atoms that cannot participate in a fulfilling path starting at an initial  $\varphi$ -atom.

To assist in the removal of useless atoms, we say that an MSCS  $S$  is *terminal* if there are no edges leading from atoms of  $S$  to atoms outside of  $S$ . For example, in the tableau of Fig. 5.3,  $\{A_7\}$  and  $\{A_5\}$  are terminal MSCS's, while  $\{A_6\}$  and  $\{A_2, A_3\}$  are not.

The following algorithm prunes the tableau by removing useless atoms.

### Algorithm PRUNE — pruning the tableau

Repeat the following steps until no further MSCS's can be removed.

- Remove an MSCS that is not reachable from an initial  $\varphi$ -atom.
- Remove a terminal MSCS that is not fulfilling.

**Example** Consider the formula  $\varphi_3$ :  $\Box \Diamond(x = 3)$ . The closure of  $\varphi_3$  is given by  $\Phi_{\varphi_3} = \Phi_{\varphi_3}^+ \cup \Phi_{\varphi_3}^-$ , where

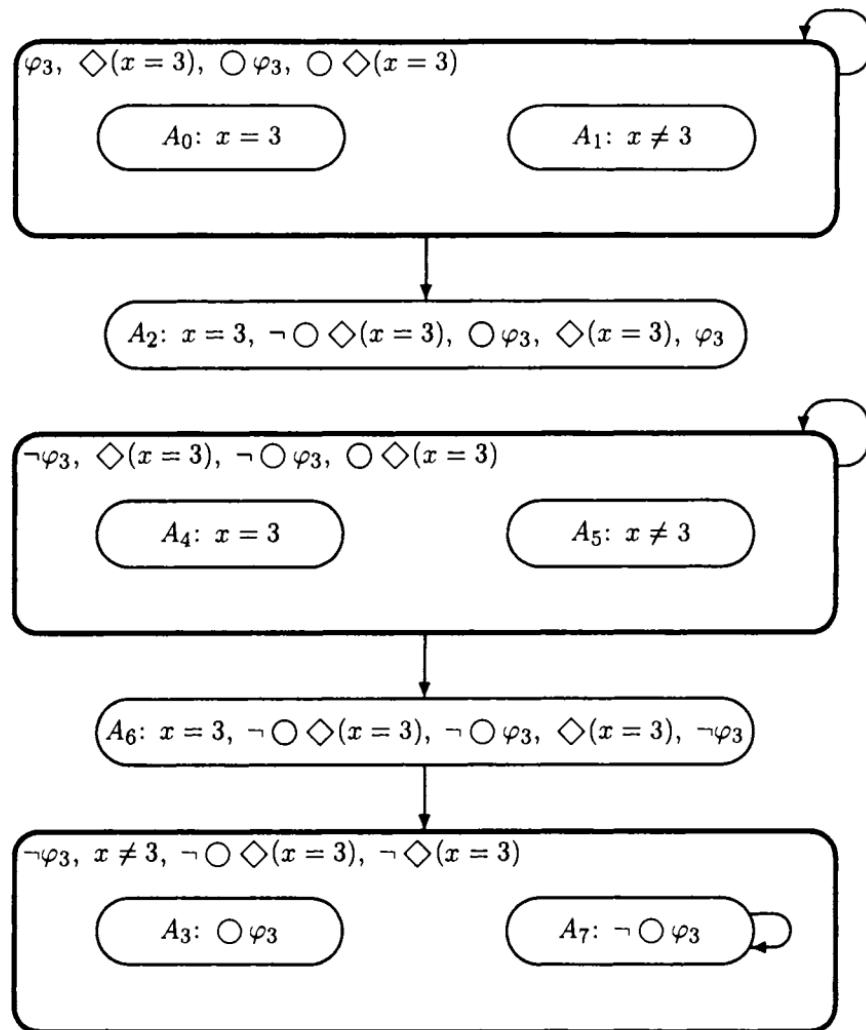
$$\Phi_{\varphi_3}^+: \{\varphi_3, \Diamond(x = 3), x = 3, \Box \Diamond(x = 3), \Box \varphi_3\}.$$

The atoms of  $\varphi_3$  are given by

$A_0:$	$\{x = 3, \Box \Diamond(x = 3), \Box \varphi_3, \Diamond(x = 3), \varphi_3\}$
$A_1:$	$\{x \neq 3, \Box \Diamond(x = 3), \Box \varphi_3, \Diamond(x = 3), \varphi_3\}$
$A_2:$	$\{x = 3, \neg \Box \Diamond(x = 3), \Box \varphi_3, \Diamond(x = 3), \varphi_3\}$
$A_3:$	$\{x \neq 3, \neg \Box \Diamond(x = 3), \Box \varphi_3, \neg \Diamond(x = 3), \neg \varphi_3\}$
$A_4:$	$\{x = 3, \Box \Diamond(x = 3), \neg \Box \varphi_3, \Diamond(x = 3), \neg \varphi_3\}$
$A_5:$	$\{x \neq 3, \Box \Diamond(x = 3), \neg \Box \varphi_3, \Diamond(x = 3), \neg \varphi_3\}$
$A_6:$	$\{x = 3, \neg \Box \Diamond(x = 3), \neg \Box \varphi_3, \Diamond(x = 3), \neg \varphi_3\}$
$A_7:$	$\{x \neq 3, \neg \Box \Diamond(x = 3), \neg \Box \varphi_3, \neg \Diamond(x = 3), \neg \varphi_3\}$

The tableau for  $\varphi_3$  is given in Fig. 5.5.

There are two MSCS's that are reachable from an initial  $\varphi_3$ -atom:  $\{A_0, A_1\}$  and  $\{A_2\}$ . The MSCS  $\{A_2\}$  is not fulfilling since it is transient. On the other hand,  $\{A_0, A_1\}$  is fulfilling because the promising formula  $\Diamond(x = 3)$  is fulfilled at  $A_0$  which contains  $x = 3$ . Consequently, pruning the tableau of Fig. 5.5, we obtain the pruned tableau of Fig. 5.6, which consists of the single  $\varphi$ -reachable MSCS  $\{A_0, A_1\}$ . Formula  $\varphi_3$  is therefore satisfiable by the model

Fig. 5.5. Tableau for formula  $\varphi_3: \Box \Diamond(x = 3)$ .

$$(\langle x: 3 \rangle, \langle x: 0 \rangle)^\omega$$

obtained by an infinite traversal of  $\{A_0, A_1\}$ . The value 0 chosen for  $x$  when visiting  $A_1$  is arbitrary; any other value can be chosen as long as it is different from 3.

If, instead of  $\varphi_3$ , we are interested in checking satisfiability of the formula  $\neg \varphi_3: \neg \Box \Diamond(x = 3)$ , we may rewrite it as  $\psi_3: \Diamond \Box(x \neq 3)$ . It is possible to use again the full tableau of Fig. 5.5, replacing  $\varphi_3$  by  $\neg \psi_3$ ,  $\bigcirc \varphi_3$  by  $\neg \bigcirc \psi_3$ ,

$\varphi_3, \diamond(x = 3), \circ\varphi_3, \circ\diamond(x = 3)$

$A_0: x = 3$

$A_1: x \neq 3$

Fig. 5.6. Pruned tableau for  $\varphi_3: \square\diamond(x = 3)$ .

$\neg\psi_3, \neg\square(x \neq 3), \neg\circ\psi_3, \neg\circ\square(x \neq 3)$

$A_0: x = 3$

$A_1: x \neq 3$

$A_2: x = 3, \circ\square(x \neq 3), \neg\circ\psi_3, \neg\square(x \neq 3), \neg\psi_3$

$\psi_3, \neg\square(x \neq 3), \circ\psi_3, \neg\circ\square(x \neq 3)$

$A_4: x = 3$

$A_5: x \neq 3$

$A_6: x = 3, \circ\square(x \neq 3), \circ\psi_3, \neg\square(x \neq 3), \psi_3$

$\psi_3, x \neq 3, \circ\square(x \neq 3), \square(x \neq 3)$

$A_3: \neg\circ\psi_3$

$A_7: \circ\psi_3$

Fig. 5.7. Tableau for  $\psi_3: \diamond\square(x \neq 3)$ .

$\Diamond(x = 3)$  by  $\neg \Box(x \neq 3)$ , and  $\bigcirc \Diamond(x = 3)$  by  $\neg \bigcirc \Box(x \neq 3)$ . This relabeled tableau is presented in Fig. 5.7.

Next, we prune the tableau, eliminating MSCS's not reachable from an initial  $\psi_3$ -atom, as well as non-fulfilling terminal MSCS's. This leads to the pruned tableau of Fig. 5.8.

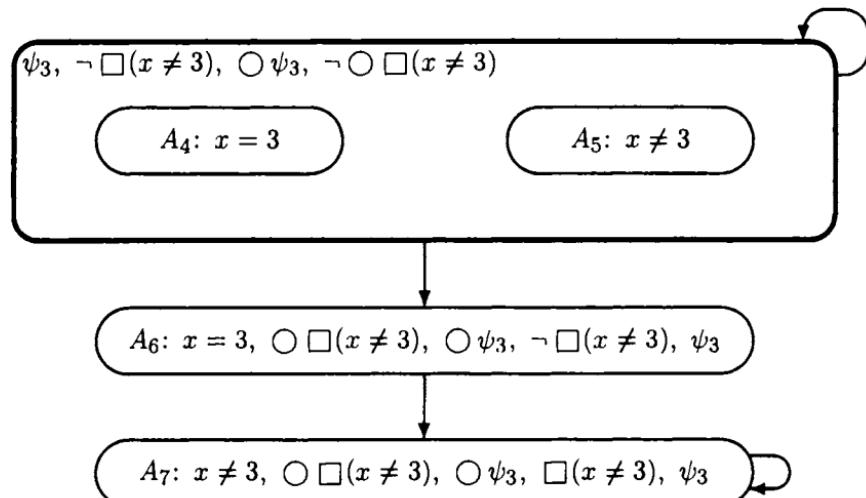


Fig. 5.8. Pruned tableau for  $\psi_3: \Diamond \Box(x \neq 3)$ .

The MSCS's of this tableau are  $\{A_4, A_5\}$ ,  $\{A_6\}$ , and  $\{A_7\}$ . Of these, the only fulfilling MSCS is  $\{A_7\}$ . Consequently, formula  $\psi_3$  is satisfiable, e.g., by the model

$$(x : 0)^\omega,$$

obtained by infinite traversal of  $\{A_7\}$ . ■

In **Problem 5.4**, the reader is requested to construct tableaux and check the satisfiability of some example formulas.

## 5.2 Satisfiability over a Finite-State Program

Algorithm SAT checks whether a given formula  $\varphi$  is satisfiable, i.e., whether there is a model  $\sigma$  such that  $\sigma \models \varphi$ . Satisfiability of a specification is a first feasibility requirement. If a specification does not pass this simple test, there is no sense in proceeding to implement the specification because it cannot be implemented.

Suppose a specification given by a formula  $\varphi$  passes the satisfiability test and a candidate implementation, represented by a finite-state program  $P$ , is proposed. The main question is whether program  $P$  satisfies the specification  $\varphi$ , i.e., whether  $\varphi$  is  $P$ -valid. We refer to this problem as the  $P$ -validity problem: Given a finite-state program  $P$  and a formula  $\varphi$ , does  $\varphi$  hold over all computations?

To tackle this problem, we first address the dual problem, called the  $P$ -satisfiability problem: Given a finite-state program  $P$  and a formula  $\varphi$ , is there a computation of  $P$  that satisfies  $\varphi$ ?

Obviously, a formula  $\varphi$  is  $P$ -valid iff  $\neg\varphi$  is not  $P$ -satisfiable. Consequently, an effective algorithm for checking  $P$ -satisfiability can be used for checking  $P$ -validity.

### Example (system LOOP)

In Fig. 5.9 we present the state-transition graph of system LOOP. The transitions of this system are the idling transition  $\tau_I$  and a transition  $\tau$  with the transition relation  $\rho_\tau$ :  $x' = (x + 1) \bmod 4$ . The justice set for LOOP is  $\mathcal{J}$ :  $\{\tau\}$ .

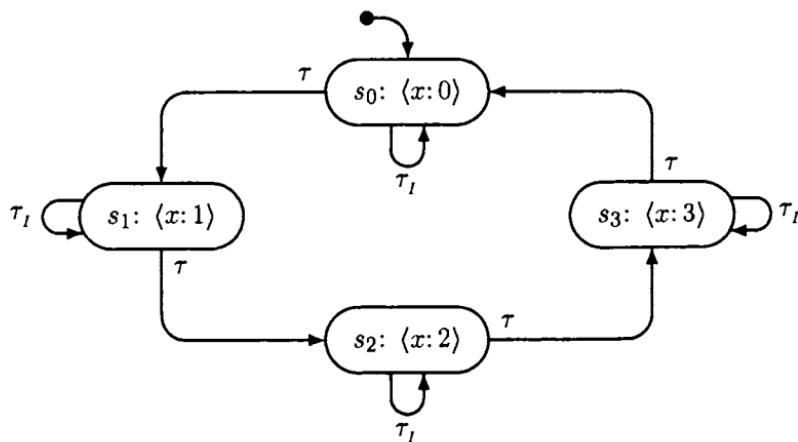


Fig. 5.9. State-transition graph of system LOOP.

We are interested in checking whether the property  $\varphi_3: \square \diamond(x = 3)$  is valid over system LOOP. Obviously, property  $\square \diamond(x = 3)$  is valid over program LOOP iff there does not exist a computation of LOOP which satisfies the negation  $\neg\varphi_3: \neg \square \diamond(x = 3)$ , which we write as  $\psi_3: \diamond \square(x \neq 3)$ . ■

The algorithm for checking the  $P$ -satisfiability of a formula  $\varphi$  strongly relies on the tableau  $T_\varphi$ , constructed for checking satisfiability of  $\varphi$ . Thus, the first step in the algorithm calls for the construction of  $T_\varphi$ . We introduce the notion of a *trail*, which relates the paths in  $T_\varphi$  and the computations of  $P$ .

For an atom  $A$ , we denote by  $\text{state}(A)$  the conjunction of all state formulas belonging to  $A$ . Recall (page 403) that  $R_{\text{sat}}$  requires that  $\text{state}(A)$  be satisfiable. An atom  $A$  is said to be *consistent with* a state  $s$  if  $s \models \text{state}(A)$ , i.e., all state formulas in  $A$  are satisfied by  $s$ .

To distinguish between paths in  $T_\varphi$  and paths in  $\mathcal{B}_{(P,\varphi)}$ , we refer to a path  $\vartheta: A_0, A_1, \dots$  in  $T_\varphi$ , such that  $A_0$  is an initial atom, as a *trail*. Let  $\vartheta: A_0, A_1, \dots$  be a trail in  $T_\varphi$  and  $\sigma: s_0, s_1, \dots$  be a computation of  $P$ . We say that the trail  $\vartheta$  is *consistent with*  $\sigma$  if  $A_j$  is consistent with  $s_j$ , for every  $j \geq 0$ .

For atom  $A \in T_\varphi$ , we denote by  $\delta(A)$  the set of successors of  $A$  in the tableau, i.e., the set of atoms  $B$  such that  $A$  is connected to  $B$ .

## The Behavior Graph

For a finite-state program  $P$  and formula  $\varphi$ , we construct the  $(P, \varphi)$ -behavior graph, denoted by  $\mathcal{B}_{(P,\varphi)}$ , which can be viewed as the cross product of  $G_P$ , the state-transition graph of program  $P$ , and the tableau  $T_\varphi$ . This structure will enable us to consider a computation  $\sigma$  of  $P$  and, at the same time, trace the trail in  $T_\varphi$  induced by  $\sigma$ . For the definition and construction of  $G_P$ , we refer the reader to Section 2.6 (page 228).

Nodes in the behavior graph are pairs  $(s, A)$ , where  $s$  is a state of  $G_P$  and  $A$  is an atom consistent with  $s$ . The graph contains a directed edge labeled by  $\tau \in \mathcal{T}$ , which connects node  $(s, A)$  to node  $(s', A')$  if and only if  $s' \in \tau(s)$  and  $A' \in \delta(A)$ .

We refer to pairs  $(s, A)$  such that  $s$  is an initial state of  $P$ ,  $A$  is an initial  $\varphi$ -atom, and  $A$  is consistent with  $s$ , as the *initial  $\varphi$ -nodes* of  $\mathcal{B}_{(P,\varphi)}$ .

Since we intend to check satisfiability of  $\varphi$  over computations, we may use the pruned version of the tableau for  $\varphi$ . Therefore,  $\delta(A)$  refers to successor atoms in the pruned tableau.

The behavior graph  $\mathcal{B}_{(P,\varphi)}$  can be constructed in an incremental manner, as described in the following algorithm:

### Algorithm BEHAVIOR-GRAF — construction of $\mathcal{B}_{(P,\varphi)}$

- Initially, place in  $\mathcal{B}_{(P,\varphi)}$  all initial  $\varphi$ -nodes. That is, all nodes of the form  $(s, A)$  where  $s$  is initial for  $P$  and  $A$  is an initial  $\varphi$ -atom consistent with  $s$ .
- Repeat the following augmentation step until no new nodes or new edges can be added to  $\mathcal{B}_{(P,\varphi)}$ :

*Augmentation Step.* Let  $(s, A)$  be a node already in  $\mathcal{B}_{(P,\varphi)}$ ,  $\tau \in \mathcal{T}$  be a transition, and  $(s', A')$  be a pair such that  $s'$  is a  $\tau$ -successor

of  $s$ ,  $A' \in \delta(A)$ , and  $A'$  is consistent with  $s'$ .

- Add  $(s', A')$  to  $\mathcal{B}_{(P, \varphi)}$  if it is not already there.
- Draw a directed edge connecting  $(s, A)$  to  $(s', A')$  labeled by  $\tau$ , if such an edge is not already in  $\mathcal{B}_{(P, \varphi)}$ .

Note that the algorithm constructs only the part of the behavior graph that is reachable from initial  $\varphi$ -nodes.

**Example** In Fig. 5.10 we present the behavior graph  $\mathcal{B}$  for system LOOP and the formula  $\psi_3$ :  $\Diamond \Box(x \neq 3)$ . This graph can be viewed as the cross product of the state-transition graph  $G_{\text{LOOP}}$  of Fig. 5.9 and the pruned tableau for  $\psi_3$ , presented in Fig. 5.8. The small arrows entering nodes  $(s_0, A_5)$  and  $(s_0, A_7)$  identify them as initial  $\psi_3$ -nodes.

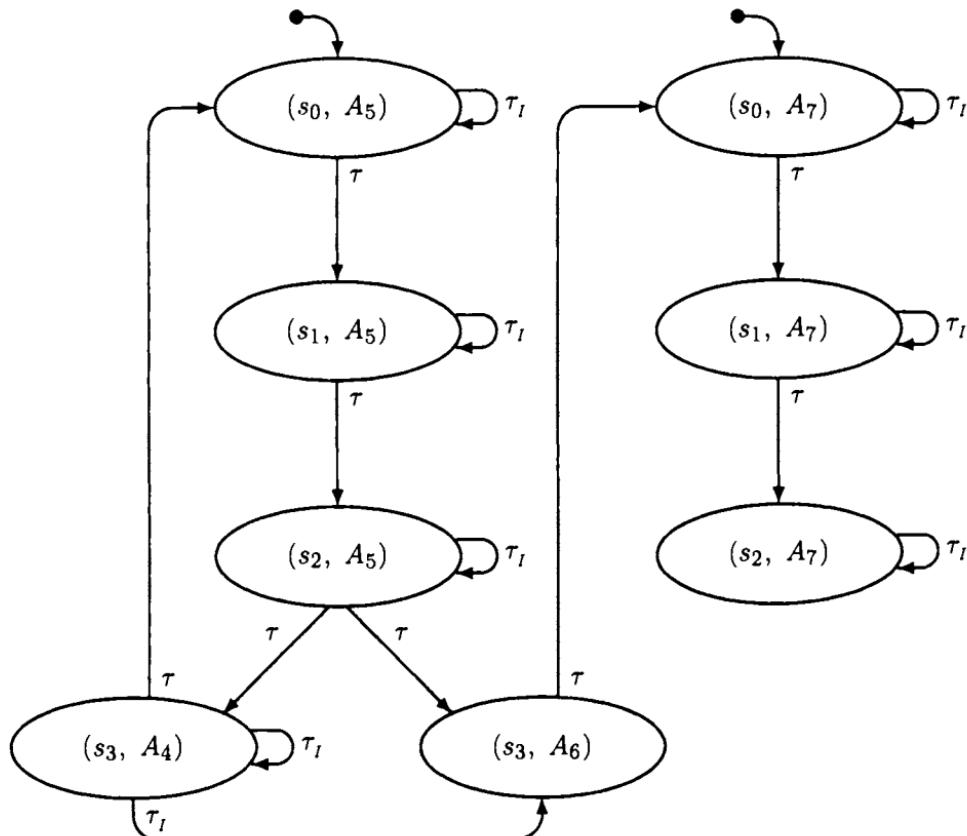


Fig. 5.10. Behavior graph for LOOP and  $\psi_3$ :  $\Diamond \Box(x \neq 3)$ .

Consider, for example, node  $(s_1, A_5)$ . In  $G_{\text{LOOP}}$ ,  $s_1$  has itself as a  $\tau_I$ -successor, and  $s_2$  as a  $\tau$ -successor. Atom  $A_5$  has in (the pruned version of)  $T_{\psi_3}$  atoms  $A_5$ ,  $A_4$ , and  $A_6$  as successors. Consequently, we may consider pairs  $(s_1, A_5)$ ,  $(s_1, A_4)$ , and  $(s_1, A_6)$  as candidates for being  $\tau_I$ -successors of  $(s_1, A_5)$  and the pairs  $(s_2, A_5)$ ,  $(s_2, A_4)$ , and  $(s_2, A_6)$ , as candidates for being  $\tau$ -successors of  $(s_1, A_5)$  in  $\mathcal{B}$ . Of these,  $(s_1, A_4)$ ,  $(s_1, A_6)$ ,  $(s_2, A_4)$ , and  $(s_2, A_6)$  are inconsistent as  $A_4$  and  $A_6$  require  $x = 3$  while the values of  $x$  in  $s_1$  and  $s_2$  are 1 and 2, respectively.

On the other hand, the  $\tau$ -successors of  $(s_2, A_5)$  must be nodes of the form  $(s_3, A_i)$  where  $A_i$  is a successor of  $A_5$  which is consistent with the condition  $x = 3$ , imposed by state  $s_3$ . The two candidates for  $A_i$  are  $A_4$  and  $A_6$  leading to  $(s_3, A_4)$  and  $(s_3, A_6)$  as successors of  $(s_2, A_5)$ .

We also note that node  $(s_3, A_6)$  has no  $\tau_I$ -successor. This is because the only  $\tau_I$ -successor of  $s_3$  is  $s_3$  itself, implying  $x = 3$ , while the only (pruned) tableau successor of  $A_6$  is  $A_7$  which requires  $x \neq 3$ .

Observe that node  $(s_2, A_7)$  has no  $\tau$ -successor. This is because the  $\tau$ -successor of  $s_2$  in program LOOP satisfies  $x = 3$ , while the only successor of  $A_7$  in the tableau is  $A_7$  itself which contains  $x \neq 3$  and is, therefore, inconsistent with state  $s_3$ . ■

### Paths in $\mathcal{B}_{(P,\varphi)}$

Let  $\pi: (s_0, A_0), (s_1, A_1), \dots$  be an infinite sequence of nodes in  $\mathcal{B}_{(P,\varphi)}$ . We say that  $\pi$  is an *initialized path* if  $(s_0, A_0)$  is an initial  $\varphi$ -node in  $\mathcal{B}_{(P,\varphi)}$ , implying  $\varphi \in A_0$ . We define  $\sigma_\pi: s_0, s_1, \dots$  and  $\vartheta_\pi: A_0, A_1, \dots$  as the state and atom sequences *induced* by  $\pi$ , respectively. Based on the construction of the behavior graph  $\mathcal{B}_{(P,\varphi)}$ , we can make the following claim:

**Claim 5.9** (graph path induces a run and a trail)

The infinite sequence  $\pi: (s_0, A_0), (s_1, A_1), \dots$ , where  $(s_0, A_0)$  is an initial  $\varphi$ -node, is a path in  $\mathcal{B}_{(P,\varphi)}$  iff  $\sigma_\pi$  is a run of  $P$  and  $\vartheta_\pi$  is a trail of  $T_\varphi$  consistent with  $\sigma_\pi$ .

Recall that a run of a fair transition system is a sequence of states that satisfies all the requirements of a computation except possibly the fairness requirements. Sequence  $\sigma$  is a run of  $P$  iff it is an infinite path in  $G_P$ .

For example, the periodic path

$$\pi: ((s_0, A_5), (s_1, A_5), (s_2, A_5), (s_3, A_4))^\omega$$

in the behavior graph of Fig. 5.10 is an initialized path which induces the run  $\sigma_\pi: (s_0, s_1, s_2, s_3)^\omega$  of LOOP and the trail  $\vartheta_\pi: (A_5, A_5, A_5, A_4)^\omega$  in  $T_{\psi_3}$ .

We recall from page 410 that a trail  $\vartheta: A_0, A_1, \dots$  (path in  $T_\varphi$ ) is called *fulfilling* if  $A_0$  is an initial atom and, for every promising formula  $\psi \in \Phi_\varphi$ ,  $\vartheta$  contains infinitely many atoms  $A_j$  which fulfill  $\psi$ . For example, the trail  $(A_5, A_5, A_5, A_4)^\omega$  in  $T_{\psi_3}$  of Fig. 5.8 is not fulfilling because it fails to fulfill  $\psi_3$ :  $\Diamond \Box(x \neq 3)$ . On the other hand, the trail  $A_7^\omega$  in the same tableau is fulfilling.

To determine whether there exists a  $P$ -computation that satisfies  $\varphi$ , it is sufficient to search for an infinite initialized path  $\pi$  in  $\mathcal{B}_{(P,\varphi)}$  such that  $\sigma_\pi$  is a fair run (and therefore a computation) and  $\vartheta_\pi$  is a fulfilling trail in  $T_\varphi$ .

### Proposition 5.10 (satisfiability and fulfilling trails)

A finite-state program  $P$  has a computation satisfying  $\varphi$  iff the behavior graph  $\mathcal{B}_{(P,\varphi)}$  contains an initialized path  $\pi$  such that  $\sigma_\pi$  is a computation of  $P$  and  $\vartheta_\pi$  is a fulfilling trail in  $T_\varphi$ .

## Adequate Subgraphs

Again, we can reduce the search for infinite paths to a search for an appropriate SCS  $S$  of  $\mathcal{B}_{(P,\varphi)}$  such that there exists a path  $\pi$  of  $\mathcal{B}_{(P,\varphi)}$  with the properties that  $\sigma_\pi$  is fair,  $\vartheta_\pi$  is fulfilling, and  $S$  consists of all the atoms that appear infinitely many times in  $\pi$ , i.e.,  $S = \text{Inf}(\pi)$ . The following criteria for fairness and fulfillment of a subgraph of  $\mathcal{B}_{(P,\varphi)}$  are derived from the criteria applied to subgraphs of  $G_P$  and  $T_\varphi$ , respectively:

Consider a behavior graph  $\mathcal{B}_{(P,\varphi)}$  for a finite-state program  $P$  and formula  $\varphi$ . A node  $(s', A')$  is defined to be a  *$\tau$ -successor* of the node  $(s, A)$  if  $\mathcal{B}_{(P,\varphi)}$  contains an edge labeled by  $\tau$  connecting  $(s, A)$  to  $(s', A')$ . Transition  $\tau$  is said to be *enabled* on node  $(s, A)$  if  $\tau$  is enabled on state  $s$ .

Let  $S \subseteq \mathcal{B}_{(P,\varphi)}$  be a subgraph of  $\mathcal{B}_{(P,\varphi)}$ . We say that transition  $\tau$  is *taken in*  $S$  if there exist two nodes  $(s, A), (s', A') \in S$  such that  $(s', A')$  is a  $\tau$ -successor of  $(s, A)$ .

Subgraph  $S$  is said to be *just* if every just transition  $\tau \in \mathcal{J}$  is either taken in  $S$  or disabled on some node in  $S$ . It is called *compassionate* if every compassionate transition is either taken in  $S$  or is disabled on all nodes belonging to  $S$ . Subgraph  $S$  is *fair* if it is both just and compassionate.

Subgraph  $S$  is called *fulfilling* if every promising formula is fulfilled by an atom  $A$  such that  $(s, A) \in S$  for some state  $s$ .

Subgraph  $S$  is called *adequate* if it is fair and fulfilling, combining the requirements that  $\sigma$  be a computation and  $\vartheta$  be a fulfilling trail. The following proposition identifies an adequate SCS as the goal of our search:

**Proposition 5.11** (satisfiability and adequate scs)

A finite-state program  $P$  has a computation satisfying  $\varphi$  iff the behavior graph  $\mathcal{B}_{(P,\varphi)}$  has an adequate SCS.

**Justification** Assume that  $P$  has a computation  $\sigma: s_0, s_1, \dots$  that satisfies  $\varphi$ . By Claims 5.2 and 5.4, there exists a fulfilling trail  $\vartheta: A_0, A_1, \dots$  consistent with  $\sigma$  such that  $A_0$  is an initial  $\varphi$ -atom.

Construct the sequence of nodes  $\pi: (s_0, A_0), (s_1, A_1), \dots$ . According to Claim 5.9,  $\pi$  is an infinite initialized path in  $\mathcal{B}_{(P,\varphi)}$ . Let  $S$  denote the subgraph  $\text{Inf}(\pi)$ , i.e., the subgraph consisting of all nodes that appear infinitely many times in  $\pi$ . It is not difficult to ascertain that  $S$  is strongly connected, fair (since  $\sigma$  is fair), and fulfilling (since  $\vartheta$  is fulfilling). It follows that  $S$  is an adequate SCS.

In the other direction, let  $S$  be an adequate SCS of  $\mathcal{B}_{(P,\varphi)}$ . Consider a path  $\pi: (s_0, A_0), (s_1, A_1), \dots$  originating at an initial node and such that  $\text{Inf}(\pi) = S$ . The path  $\pi$  can be chosen so that, for every two nodes  $n, n' \in S$  such that there is a  $\mathcal{B}_{(P,\varphi)}$ -edge connecting  $n$  to  $n'$ , there are infinitely many positions  $j$  such that  $(s_j, A_j) = n$  and  $(s_{j+1}, A_{j+1}) = n'$ . Thus,  $\pi$  not only visits every  $n \in S$  infinitely many times, but also traces every edge contained in  $S$  infinitely many times. It can be established that  $\sigma_\pi: s_0, s_1, \dots$  is a computation of  $P$  and  $\vartheta_\pi: A_0, A_1, \dots$  is a fulfilling trail over  $\sigma_\pi$  such that  $\varphi \in A_0$ . Thus,  $\sigma_\pi$  is a computation of program  $P$  which satisfies the formula  $\varphi$ . ■

**Example** The behavior graph for system **LOOP** and formula  $\psi_3$ , presented in Fig. 5.10, has no adequate subgraphs. To see this, we may examine each of the MSCS's of this behavior graph and show that they cannot contain adequate subgraphs.

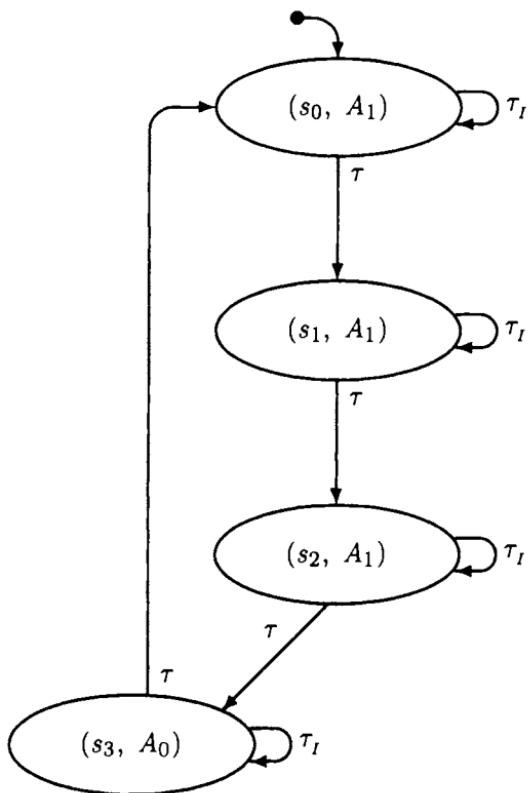
- The MSCS  $\{(s_0, A_5), (s_1, A_5), (s_2, A_5), (s_3, A_4)\}$  is fair but not fulfilling. It is fair because  $\tau$ , the only just transition in system **LOOP**, is taken. It is not fulfilling because atoms  $A_4$  and  $A_5$  contain the formula  $\psi_3: \Diamond \Box(x \neq 3)$ , which promises  $\Box(x \neq 3)$ , but neither  $A_4$  nor  $A_5$  contain the promised  $\Box(x \neq 3)$ . Therefore, this MSCS cannot contain any fulfilling subgraph.
- Each of the singleton subgraphs  $\{(s_0, A_7)\}$ ,  $\{(s_1, A_7)\}$ , and  $\{(s_2, A_7)\}$  is fulfilling but not fair. In particular, they are not just with respect to transition  $\tau$ , since  $\tau$  is enabled on all their nodes but not taken.
- The subgraph  $\{(s_3, A_6)\}$  is neither fair (it is unjust towards  $\tau$ ) nor fulfilling (being transient).

Since no subgraph is adequate, we conclude by Proposition 5.11 that **LOOP** has no computation that satisfies  $\psi_3: \Diamond \Box(x \neq 3)$ .

To illustrate the case that the behavior graph contains an adequate SCS, consider formula  $\varphi_3: \Box \Diamond(x = 3)$  and its pruned tableau  $T_{\varphi_3}$ , presented in

Fig. 5.6.

In Fig. 5.11 we present the behavior graph for system LOOP and formula  $\varphi_3$ .

Fig. 5.11. Behavior graph for LOOP and  $\varphi_3$ :  $\square \diamond(x = 3)$ .

Among other scs's, the graph of Fig. 5.11 contains the adequate subgraph

$$S: \{(s_0, A_1), (s_1, A_1), (s_2, A_1), (s_3, A_0)\}$$

Subgraph  $S$  is fair since transition  $\tau$  is taken in it. It is fulfilling because atom  $A_0$ , containing the  $\Phi_{\varphi_3}$ -formula  $x = 3$ , fulfills  $\diamond(x = 3)$ , the only promising formula of  $\Phi_{\varphi_3}$ .

We conclude by Proposition 5.11 that system LOOP has a computation satisfying  $\varphi_3$ :  $\square \diamond(x = 3)$ . Indeed, the periodic computation

$$\sigma: (s_0, s_1, s_2, s_3)^\omega,$$

traversing  $S$ , satisfies  $\varphi_3$ ; It induces the fulfilling trail

$\vartheta: (A_1, A_1, A_1, A_0)^\omega$

in  $T_{\varphi_3}$ . ■

It remains to present an algorithm for checking whether a given subgraph  $S$  contains an adequate SCS. Without loss of generality, we may assume that  $S$  is already an SCS.

Unlike the situation with Algorithm SAT which only requires that  $S$  be fulfilling, it is no longer sufficient to consider only maximal SCS's. While the properties of fulfillment and justice extend from an SCS  $S$  to any SCS containing  $S$ , the property of being compassionate is not monotonic. Consider, for example, the fragment of behavior graph presented in Fig. 5.12, where the compassionate transition  $\tau_2$  is enabled on  $s_5$  but disabled on  $s_4$ . Obviously, SCS  $\{(s_4, A_7)\}$  is compassionate because  $\tau_2$  is disabled on all the states occurring in this subgraph. On the other hand, the larger SCS  $\{(s_4, A_7), (s_5, A_8)\}$  is not compassionate because  $\tau_2$  is enabled on  $(s_5, A_8)$  but not taken in the subgraph.

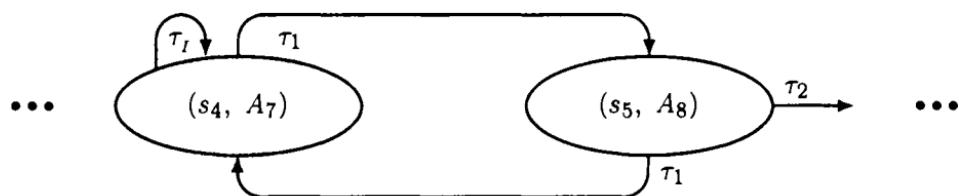


Fig. 5.12. A fragment of a behavior graph.

It follows that an SCS  $S$  may not be adequate by itself but may contain an adequate subgraph  $S' \subseteq S$ .

The following Algorithm ADEQUATE-SUB accepts as input an SCS  $S$  and returns as output an SCS  $S' \subseteq S$ . If  $S'$  is empty, this means that  $S$  contains no adequate subgraph. Otherwise,  $S'$  is an adequate SCS contained in  $S$ . The algorithm uses the notation  $EN(T, S)$  to denote the set of all nodes in  $S$  on which some transition in  $T \subseteq \mathcal{T}$  is enabled.

**Algorithm ADEQUATE-SUB** — check for adequate subgraphs

**recursive function** *adequate-sub*( $S$ : SCS) **returns** SCS

<b>if</b> $S$ is not fulfilling <b>then return</b> $\emptyset$	-- failure
<b>if</b> $S$ is not just <b>then return</b> $\emptyset$	-- failure
<b>if</b> $S$ is compassionate <b>then return</b> $S$	-- success

- -  $S$  is fulfilling and just but not compassionate. Let  $T \subseteq C$  be the set of all compassionate transitions that are not taken in  $S$ .
- - Clearly,  $EN(T, S) \neq \emptyset$ .

**let**  $U = S - EN(T, S)$ .

Decompose  $U$  into MSCS's  $U_1, \dots, U_k$ .

**let**  $V = \emptyset, i = 1$

**while**  $V = \emptyset$  and  $i \leq k$  **do**

**let**  $V = \text{adequate-sub}(U_i)$ .

$i := i + 1$

**end-while**

**return**  $V$

The algorithm consists of the recursive procedure *adequate-sub*. The first two steps check whether  $S$  itself is fulfilling and just. Since these two properties are monotonic, if  $S$  is non-fulfilling or unjust we may safely conclude that neither  $S$  nor any of its subgraphs can be adequate. Consequently, the procedure returns  $\emptyset$ , denoting a failure to locate an adequate subgraph. Thus, we proceed in the case that  $S$  is fulfilling and just. If  $S$  itself is compassionate then it is clearly adequate and the procedure returns  $S$  as a result.

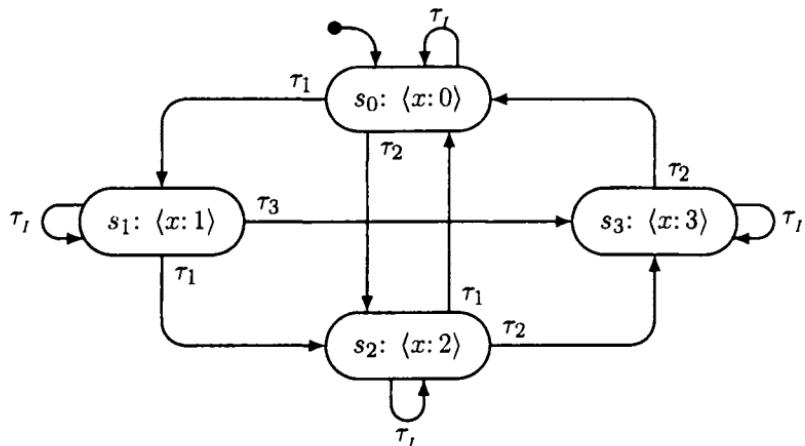
The rest of the procedure deals with the case that  $S$  is fulfilling and just but not compassionate. In that case there are some compassionate transitions that are not taken in  $S$  but are enabled on some nodes in  $S$ . Since none of these transitions can be taken in any subgraph of  $S$ , the only way to achieve compassion is by removing from  $S$  the nodes on which untaken compassionate transitions are enabled.

Such a removal may yield a graph  $U$  that is no longer strongly connected. Therefore, we decompose  $U$  into MSCS's  $U_1, \dots, U_k$ , by Algorithm DECOMPOSE. Next, we recursively apply *adequate-sub* to each  $U_i, i = 1, \dots, k$ . If any of these applications yields a non-empty result  $V$ , we return the same  $V \subseteq S$  as an adequate subgraph. If all applications yield the empty set, we conclude that  $S$  does not contain an adequate SCS and return  $\emptyset$  as the final result.

**Examples** To illustrate a case in which the algorithm returns an adequate SCS, consider system  $\text{LOOP}^+$  presented in Fig. 5.13. System  $\text{LOOP}^+$  differs from system  $\text{LOOP}$  by having three transitions, leading to additional succession relations between the states. The justice and compassion sets for system  $\text{LOOP}^+$  are

$$\mathcal{J}: \{\tau_1, \tau_2\} \quad \text{and} \quad \mathcal{C}: \{\tau_3\}.$$

We just established that system  $\text{LOOP}$  does not have a computation which satisfies  $\psi_3: \Diamond \Box(x \neq 3)$  and therefore system  $\text{LOOP}$  satisfies the formula  $\varphi_3: \Box \Diamond(x = 3)$ .

Fig. 5.13. System  $\text{LOOP}^+$ .

Let us ask the same question about system  $\text{LOOP}^+$ . In Fig. 5.14, we present the behavior graph for  $\text{LOOP}^+$  and  $\psi_3: \Diamond \Box(x \neq 3)$  using the pruned tableau for  $\psi_3$  presented in Fig. 5.8. Compared to the previous analysis of  $\text{LOOP}$ , there is an additional MSCS, given by

$$S: \{(s_0, A_7), (s_1, A_7), (s_2, A_7)\}.$$

Function *adequate-sub*, applied to  $S$ , finds that it is fulfilling and just but not compassionate. It is fulfilling because the only promising formula of  $A_7$  is the formula  $\psi_3: \Diamond \Box(x \neq 3)$  which is fulfilled by  $\Box(x \neq 3) \in A_7$ . It is just because the two just transitions,  $\tau_1$  and  $\tau_2$ , are taken in  $S$ . It is not compassionate because transition  $\tau_3$  is enabled on  $(s_1, A_7)$  but is not taken in  $S$ . Consequently, the function constructs  $U: \{(s_0, A_7), (s_2, A_7)\}$  by removing  $(s_1, A_7)$  from  $S$ . Subgraph  $U$  is already strongly connected, so no decomposition is required. It is straightforward to check that  $U$  is adequate. Subgraph  $U$  is fulfilling because  $A_7$  fulfills all that it promises. It is fair because the only transitions enabled on either  $s_0$  or  $s_2$  are  $\tau_1$  and  $\tau_2$  which are both taken in  $U$ .

We conclude that system  $\text{LOOP}^+$  has a computation satisfying  $\psi_3: \Diamond \Box(x \neq 3)$ . Indeed,  $\sigma: (s_0, s_2)^\omega$  is a computation of  $\text{LOOP}^+$  satisfying  $\psi_3: \Diamond \Box(x \neq 3)$ . Therefore,  $\varphi_3$  is not  $P$ -valid over system  $\text{LOOP}^+$ . ■

## *P*-Satisfiability

We can now formulate the main algorithm which checks whether a finite-state program  $P$  has a computation satisfying a formula  $\varphi$ .

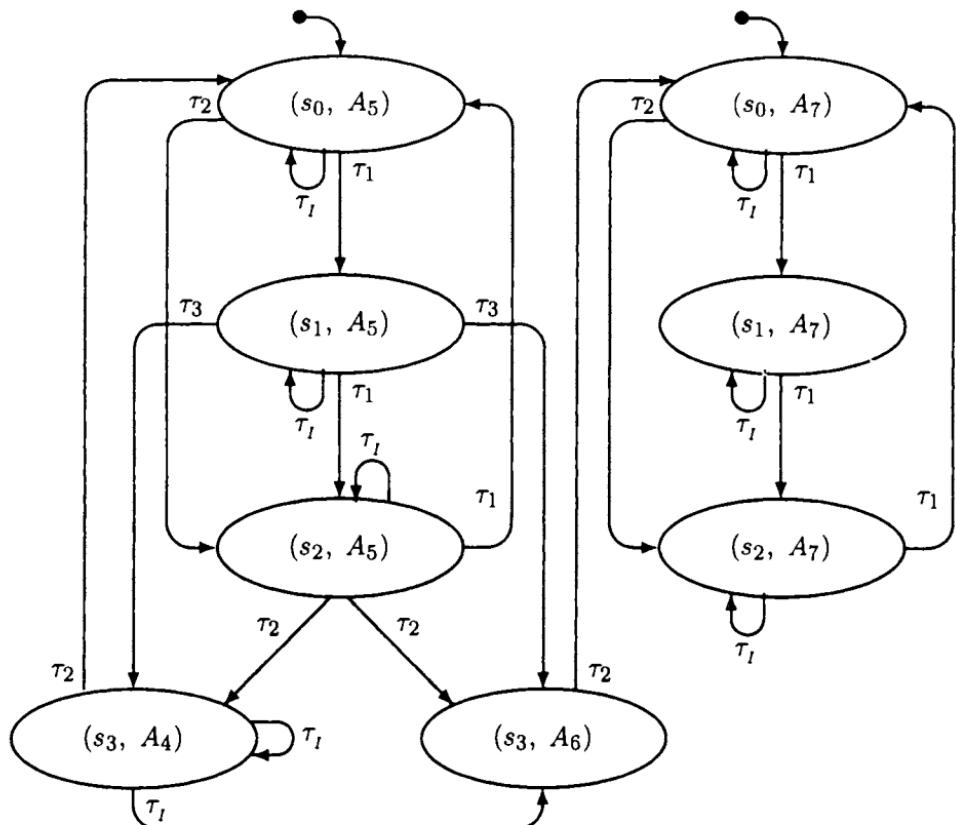


Fig. 5.14. Behavior graph for  $\text{LOOP}^+$  and  $\psi_3$ :  $\Diamond \Box(x \neq 3)$ .

### Algorithm P-SAT — satisfiability of a formula over a program

To check whether finite-state program  $P$  has a computation that satisfies the temporal formula  $\varphi$ , perform the following steps:

- Construct the state-transition graph  $G_P$ .
- Construct the pruned tableau  $T_\varphi$ .
- Construct the behavior graph  $\mathcal{B}_{(P,\varphi)}$ .
- Decompose  $\mathcal{B}_{(P,\varphi)}$  into MSCS's  $S_1, \dots, S_t$ .
- For each  $i = 1, \dots, t$ , apply Algorithm ADEQUATE-SUB to  $S_i$ .  
If any of these applications returns a nonempty result,  $P$  has a computation satisfying  $\varphi$ . This computation can be constructed by forming a path  $\pi$  that leads from an initial node to the returned adequate subgraph

$S$ , and then continues to visit each node of  $S$  infinitely many times. The desired computation is the computation  $\sigma_\pi$  induced by  $\pi$ .

If all applications return the empty set as result,  $P$  has no computation satisfying  $\varphi$ .

In **Problem 5.5**, the reader is requested to use Algorithm P-SAT in order to check whether the formula  $\square \diamond(x = 1) \wedge \square(x \neq 3)$  is satisfiable over system LOOP<sup>+</sup>.

### 5.3 Validity over a Finite-State Program: Examples

Similar to Algorithm SAT, Algorithm P-SAT can be used to check validity of a property over a finite-state program  $P$ . Let  $\varphi$  be a temporal formula specifying the property of interest. Apply Algorithm P-SAT to check whether program  $P$  has a computation satisfying  $\neg\varphi$ . If  $P$  has a computation  $\sigma$  satisfying  $\neg\varphi$  then  $\varphi$  is not  $P$ -valid. If the algorithm declares that  $P$  has no computation satisfying  $\neg\varphi$ , then  $\varphi$  is  $P$ -valid.

We will present several examples illustrating this process.

**Example** (checking accessibility for MUX-SEM)

In Section 2.6 we considered finite-state program MUX-SEM, presented in Fig. 5.15 (see also Fig. 1.6). We verified that this program satisfies the safety property of mutual exclusion specified by the formula

$$\square \neg(at\_l_3 \wedge at\_m_3).$$

local  $y$ : integer where  $y = 1$

$$P_1 :: \left[ \begin{array}{l} l_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} l_1: \text{noncritical} \\ l_2: \text{request } y \\ l_3: \text{critical} \\ l_4: \text{release } y \end{array} \right] \end{array} \right] \parallel P_2 :: \left[ \begin{array}{l} m_0: \text{loop forever do} \\ \quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: \text{request } y \\ m_3: \text{critical} \\ m_4: \text{release } y \end{array} \right] \end{array} \right]$$

Fig. 5.15. Program MUX-SEM (mutual exclusion by semaphores).

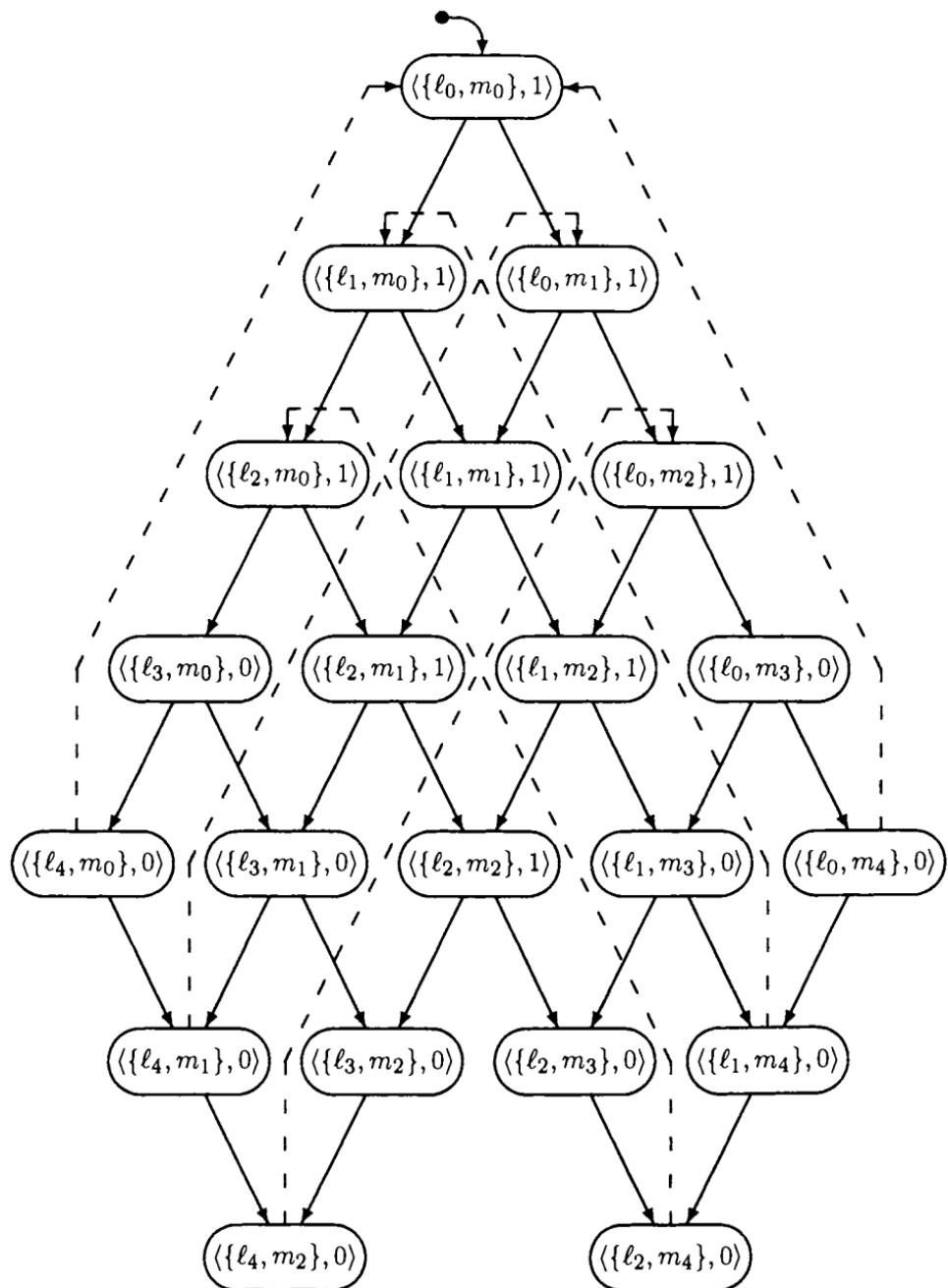


Fig. 5.16. State-transition graph for MUX-SEM.

Here, we wish to establish that program MUX-SEM satisfies the response property

$$\varphi: \text{at\_}\ell_2 \Rightarrow \Diamond \text{at\_}\ell_3.$$

In Fig. 5.16 we duplicate the state-transition graph for program MUX-SEM. This graph was previously presented in Fig. 2.27 (page 230).

In Fig. 5.17 we present a pruned tableau  $T_\psi$  for the formula  $\neg\varphi$ , which can be written as

$$\psi: \Diamond \left( \underbrace{\text{at\_}\ell_2 \wedge \Box \neg \text{at\_}\ell_3}_p \right)$$

---

$\Box \neg \text{at\_}\ell_3, \bigcirc \Box \neg \text{at\_}\ell_3, \neg \text{at\_}\ell_3$

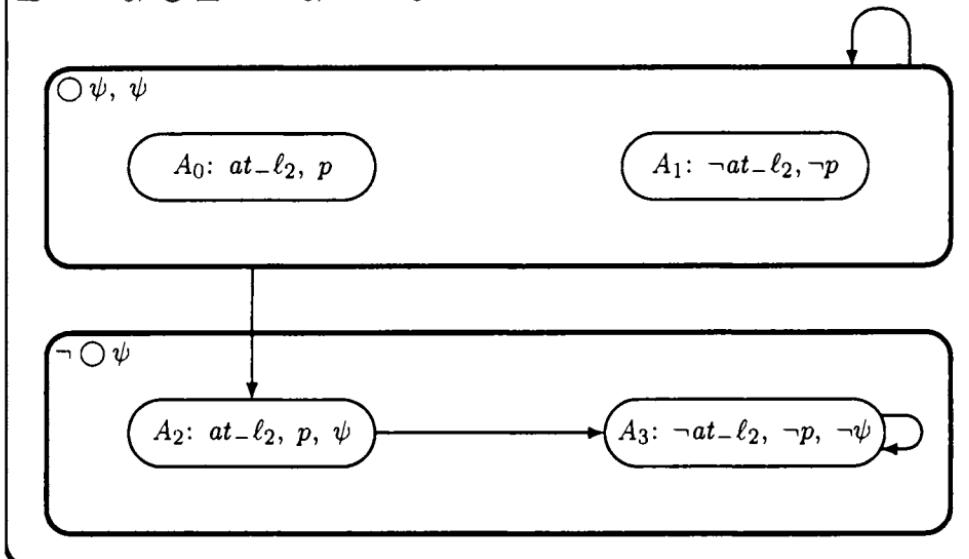


Fig. 5.17. Pruned and optimized tableau for  $\psi: \Diamond(\text{at\_}\ell_2 \wedge \Box \neg \text{at\_}\ell_3)$ .

In the construction of the tableau of Fig. 5.17, we employed an additional optimization. It can be seen that any atom that participates in a fulfilling subgraph in a tableau for a formula of the form  $\psi: \Diamond p$  must be reachable not only from an atom containing  $\psi$  but also from an atom containing  $p$ .

To see this, let  $A$  be an atom participating in a fulfilling SCS  $S$ , such that  $A$  is reachable from the initial  $\psi$ -atom  $A_0$ . We consider two cases. First, assume that  $\psi \in A$ . Since the SCS  $S$  is fulfilling, there must exist an atom  $B \in S$ , such that  $p \in B$ . As  $S$  is strongly connected and non-transient, there exists a path

from  $B$  to  $A$ . Thus,  $A$  is reachable from an atom containing  $p$ . For the other case, assume that  $\psi \notin A$ . In that case, we have a path

$$A_0, A_1, \dots, A_k = A,$$

where  $\psi = \diamond p \in A_0$  but  $\psi \notin A_k = A$ . Let  $A_i$  be the last atom in that path such that  $\psi \in A_i$ , implying  $\psi \notin A_{i+1}$ . It is not difficult to see that  $p \in A_i$  and, therefore,  $A$  is reachable from an atom containing  $p$ .

In our case,  $p$  is  $\text{at-}\ell_2 \wedge \square \neg \text{at-}\ell_3$ . It follows that all atoms reachable from an atom containing  $\psi$ :  $\diamond p$  must also be reachable from an atom containing the formula  $p$ :  $\text{at-}\ell_2 \wedge \square \neg \text{at-}\ell_3$  and must therefore contain the formula  $\square \neg \text{at-}\ell_3$  themselves.

It is clear that the fulfilling SCS's in the tableau of Fig. 5.17 are all the SCS's that contain either  $A_0$  or  $A_3$ . Atom  $A_2$  cannot participate in a fulfilling SCS, even though it fulfills  $\psi$  and  $\neg \square \neg \text{at-}\ell_3$ , the only promising formulas in  $\Phi_\psi$ . This is because the only SCS containing  $A_2$  is the transient singleton SCS  $\{A_2\}$ .

In Fig. 5.18 we present a fragment of the behavior graph for program MUX-SEM and formula  $\psi$ . It is sufficient to examine this fragment since this is the only part of the graph that is reachable from initial nodes and contains nodes that can participate in fulfilling subgraphs, i.e., nodes of the form  $(s, A_0)$  or  $(s, A_3)$  for some state  $s$  of program MUX-SEM. The full representation of this graph should have included edges, labeled by  $\tau_i$ , connecting each node to itself. To make the presentation of the graph less cluttered, we omit all these self-connecting edges.

Note, in particular, that none of the nodes associated with location  $\ell_2$  and atom  $A_2$  has any successor. This is because, in program MUX-SEM, all successors of a state satisfying  $\text{at-}\ell_2$  must satisfy  $\text{at-}\ell_2 \vee \text{at-}\ell_3$  while the only successor of  $A_2$  in the tableau of Fig. 5.16 is  $A_3$  which requires  $\neg \text{at-}\ell_2$  and  $\neg \text{at-}\ell_3$ .

Application of algorithm P-SAT to the complete graph identifies two MSCS's. The first MSCS consists of five nodes associated with location  $\ell_1$  and atom  $A_1$ . This five-node MSCS is unjust towards transition  $\ell_1$  which is enabled on all of its states but not taken within the MSCS. The second MSCS is the subgraph

$$S: \{b_0, b_1, b_2, b_3, b_4\}.$$

However, subgraph  $S$  is unfair. In particular, it is unfair towards compassionate transition  $\ell_2$  which is enabled on nodes  $b_0, b_1, b_2$  in  $S$  but not taken in  $S$ . The next step of algorithm ADEQUATE-SUB is to remove these three nodes from  $S$ , returning the result  $U = \{b_3, b_4\}$ . Decomposing  $U$  into MSCS's we obtain the following two singleton subgraphs

$$U_1: \{b_3\} \quad \text{and} \quad U_2: \{b_4\}.$$

Further analysis of these subgraphs yields that they are unjust. Subgraph  $U_1$  is unjust towards  $m_3$ , while subgraph  $U_2$  is unjust towards  $m_4$ . The main algorithm will therefore return the result  $\emptyset$ , leading to the conclusion that the behavior

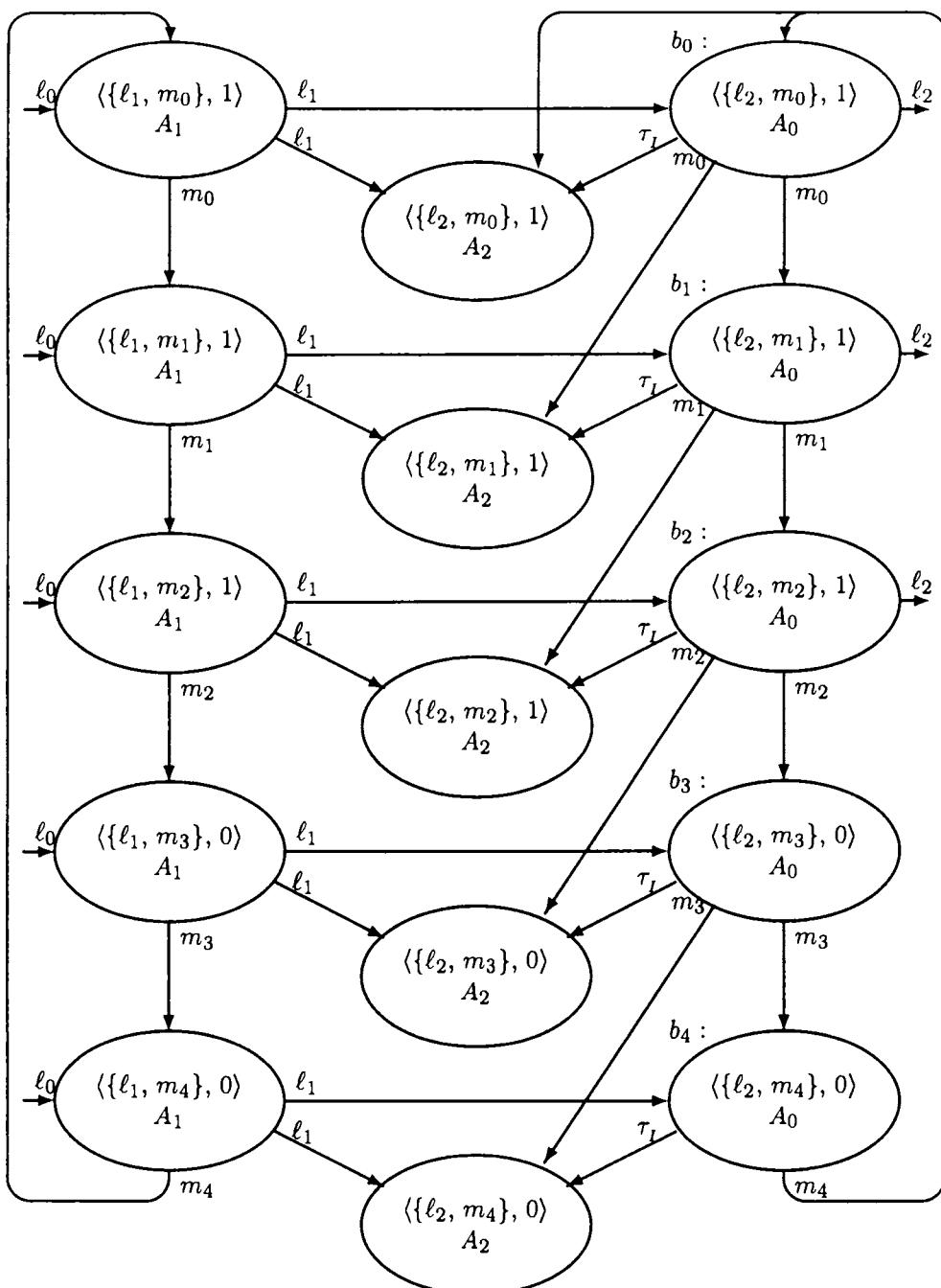


Fig. 5.18. Fragment of the behavior graph for MUX-SEM and  $\psi: \diamondsuit(at\_l_2 \wedge \square \neg at\_l_3)$ .

graph  $\mathcal{B}_{(\text{MUX-SEM}, \psi)}$  does not contain an adequate SCS. We conclude that there is no computation of program MUX-SEM which satisfies  $\psi: \diamond(at\_l_2 \wedge \square \neg at\_l_3)$ . Thus, all computations of MUX-SEM satisfy  $\varphi: at\_l_2 \Rightarrow \diamond at\_l_3$  and hence  $\varphi$  is valid over MUX-SEM. ■

### Example (checking accessibility for MUX-PET1)

Next, we consider program MUX-PET1 presented in Fig. 5.19 (see also Fig. 1.13). The state-transition graph for this program was presented in Fig. 2.28 and is duplicated in Fig. 5.20.

```
local y1, y2: boolean where y1 = F, y2 = F
      s      : integer where s = 1
```

$P_1 ::$	$\ell_0: \text{loop forever do}$ $\quad \left[ \begin{array}{l} \ell_1: \text{noncritical} \\ \ell_2: (y_1, s) := (\text{T}, 1) \\ \ell_3: \text{await } \neg y_2 \vee s \neq 1 \\ \ell_4: \text{critical} \\ \ell_5: y_1 := \text{F} \end{array} \right]$
$P_2 ::$	$m_0: \text{loop forever do}$ $\quad \left[ \begin{array}{l} m_1: \text{noncritical} \\ m_2: (y_2, s) := (\text{T}, 2) \\ m_3: \text{await } \neg y_1 \vee s \neq 2 \\ m_4: \text{critical} \\ m_5: y_2 := \text{F} \end{array} \right]$

Fig. 5.19. Program MUX-PET1 (Peterson's algorithm for mutual exclusion) — version 1.

For this program, we wish to establish the accessibility property

$$\varphi: at\_l_2 \Rightarrow \diamond at\_l_4.$$

A pruned and optimized tableau for the formula

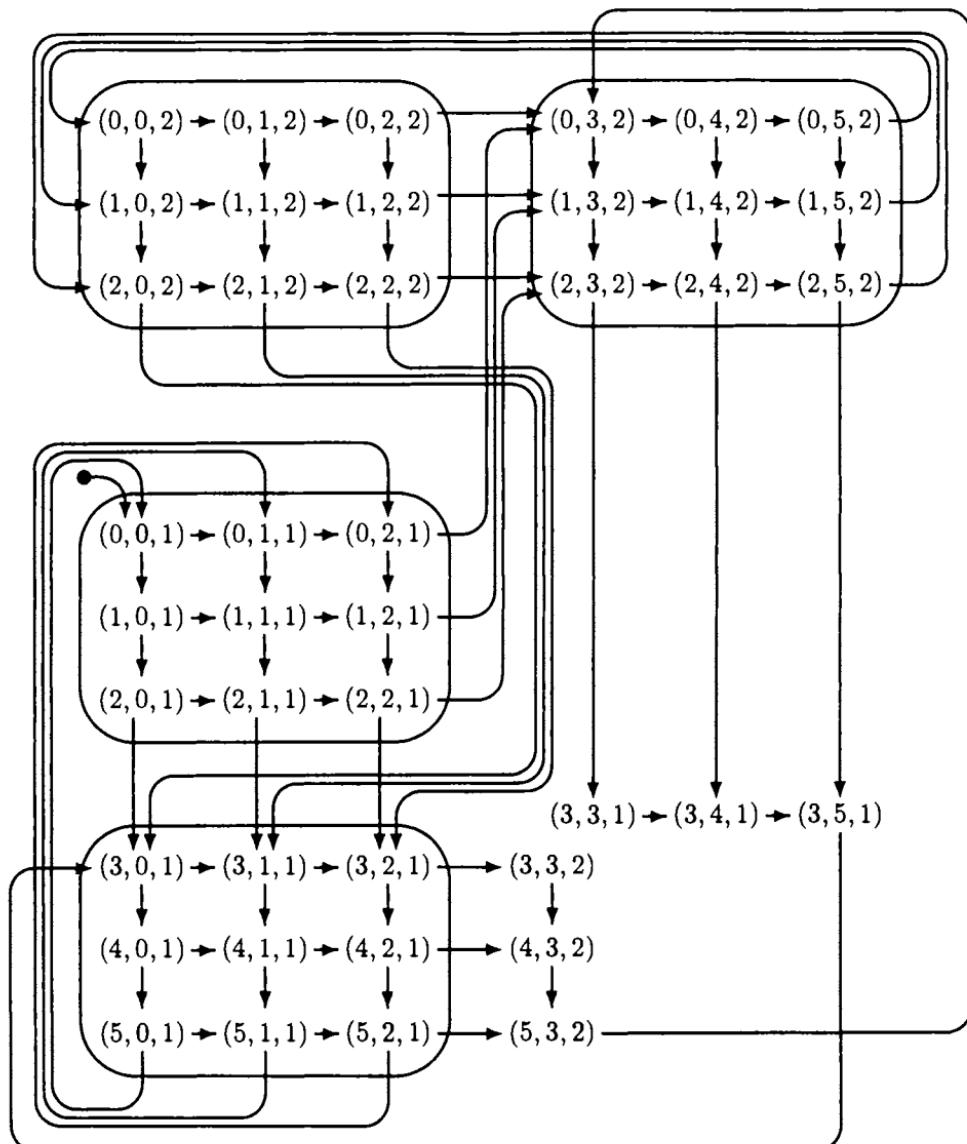


Fig. 5.20. State-transition diagram for MUX-PET1.

$$\psi: \quad \Diamond \left( \underbrace{at_{\ell_2} \wedge \square \neg at_{\ell_4}}_p \right),$$

which is congruent to  $\neg\varphi$ , is presented in Fig. 5.21. This tableau is similar to the pruned tableau of Fig. 5.17 for  $\Diamond(at_{\ell_2} \wedge \square \neg at_{\ell_3})$ , except that  $at_{\ell_3}$  is

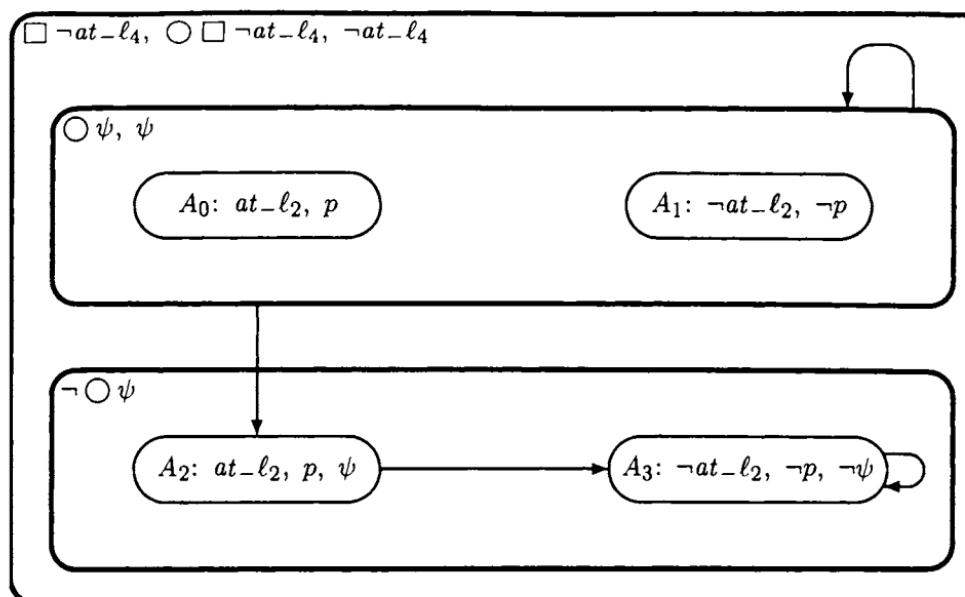


Fig. 5.21. Pruned and optimized tableau for  $\psi: \diamond(at_{\ell_2} \wedge \square \neg at_{\ell_4})$ .

replaced by  $at_{\ell_4}$ .

In Fig. 5.22 we present the relevant fragment of the behavior graph for MUX-PET1 and  $\psi$  which contains all the nodes reachable from nodes containing  $\psi$ . Each node in this graph includes a triple of the form  $\langle i, j, k \rangle$  representing a state in which  $\pi: \{\ell_i, m_j\}$  and  $s: k$ .

Decomposition of the graph of Fig. 5.22 yields the following MSCS's:

- $S_1: \{c_0, c_1, c_2, c_3, c_4, c_5\},$
- $S_2: \{b_3\}, S_3: \{b_4\}, S_4: \{b_5\}, S_5: \{b_0\}, S_6: \{b_1\}, S_7: \{b_2\},$
- $S_8: \{a_3\},$
- $S_9: \{d_3\}, S_{10}: \{d_4\}, S_{11}: \{d_5\}, S_{12}: \{d_0\}, S_{13}: \{d_1\}, S_{14}: \{d_2\},$
- $S_{15}: \{e_3\}, S_{16}: \{e_4\}, S_{17}: \{e_5\}, S_{18}: \{e_0\}, S_{19}: \{e_1\}, S_{20}: \{e_2\},$
- $S_{21}: \{f_3\}.$

None of these subgraphs is just. Subgraphs  $S_1$  and  $S_9-S_{14}$  are unjust towards transition  $\ell_2$ . Subgraphs  $S_5-S_8$  and  $S_{18}-S_{21}$  are unjust towards  $\ell_3$ . Subgraphs  $S_2$  and  $S_{15}$  are unjust towards  $m_3$ . Subgraphs  $S_3$  and  $S_{16}$  are unjust towards  $m_4$ . Finally, subgraphs  $S_4$  and  $S_{17}$  are unjust towards  $m_5$ . Thus, none of the computations of MUX-PET1 satisfies  $\psi: \diamond(at_{\ell_2} \wedge \square \neg at_{\ell_4})$ .

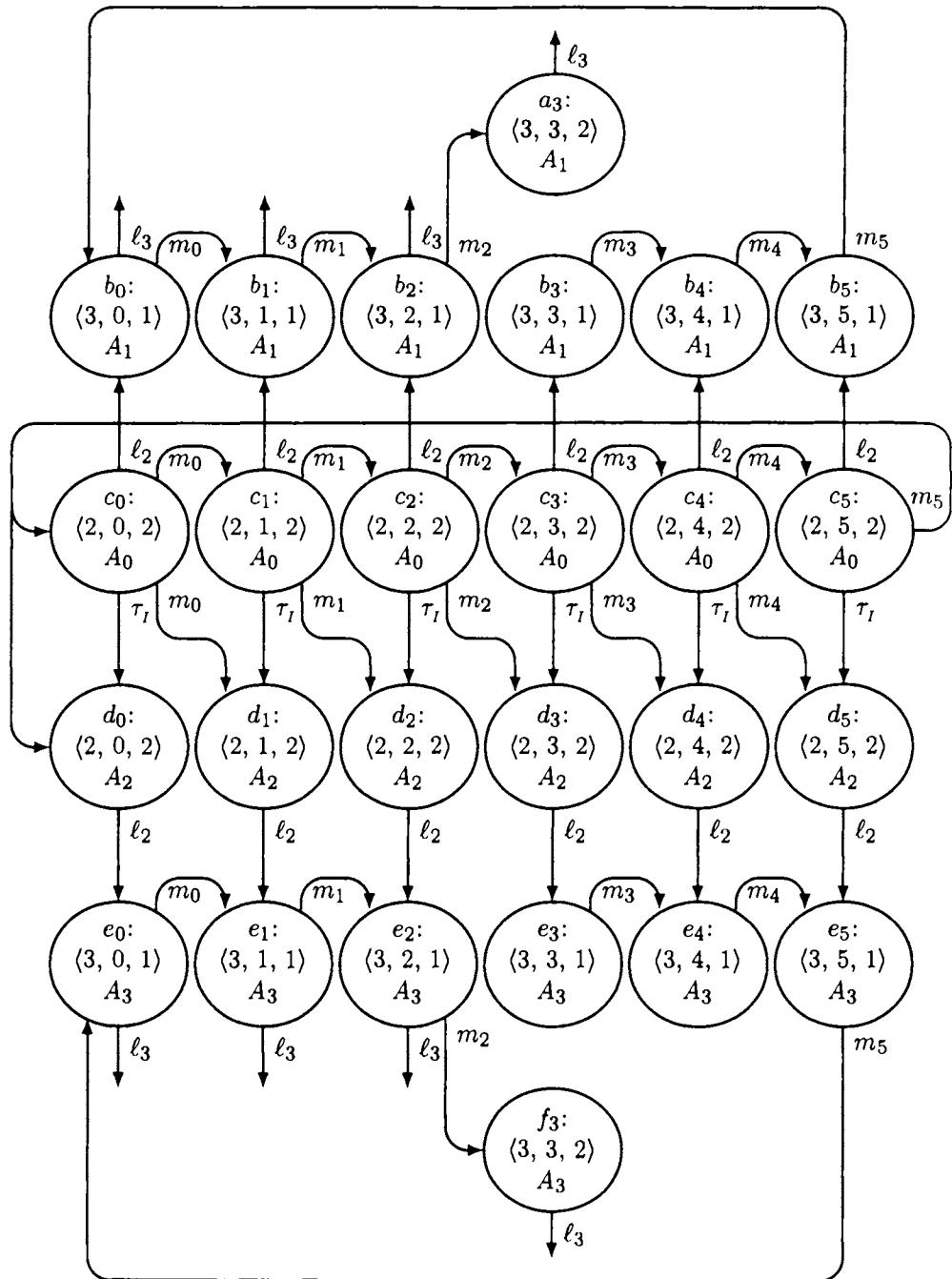


Fig. 5.22. Fragment of the behavior graph for MUX-PETI and  $\psi: \Diamond(at\_\ell_2 \wedge \Box \neg at\_\ell_4)$ .

We conclude that the accessibility property

$$\varphi: \text{at\_}\ell_2 \Rightarrow \Diamond \text{at\_}\ell_4$$

is valid over program MUX-PET1. ■

In **Problem 5.6**, the reader is requested to use the methods of this section to check whether either of the formulas  $\text{at\_}\ell_3 \wedge \text{at\_}m_2 \Rightarrow (\neg \text{at\_}m_4) \mathcal{U} (\text{at\_}\ell_4)$  and  $\text{at\_}\ell_4 \wedge \text{at\_}m_3 \Rightarrow (\text{at\_}\ell_{3,4}) \mathcal{S} (\neg \text{at\_}m_3)$  is valid over Program MUX-PET1.

## 5.4 Incremental Tableau Construction

As was observed in some of the preceding examples, the full tableau is often large, even for simple formulas. We suggested a process of pruning the tableau *after* it was constructed and also proposed several heuristics for considering only part of the tableau. In the rest of this chapter we study several possible improvements to tableau construction. In this section, we discuss incremental tableau construction which will increase the efficiency of the preceding algorithms.

Obviously, it is not necessary to include in the tableau atoms that are not reachable from an initial  $\varphi$ -atom. This observation was used in the pruning process in order to remove unreachable atoms after construction of the complete tableau. A more efficient approach will attempt to construct only  $\varphi$ -reachable atoms. This leads to the process of *incremental tableau construction* which starts with initial  $\varphi$ -atoms and only constructs atoms that are reachable from previously constructed atoms.

### Computing the Cover of a Formula Set

An important component of the incremental algorithm is a procedure for constructing all the atoms that contain a given set of formulas  $B$ . We refer to this set of atoms as the *cover* of  $B$ .

Referring to the  $\alpha$ -tables of Figures 5.1 and 5.2, we say that a set of formulas  $B$  is  $\alpha$ -closed if, for every  $\alpha$ -formula  $r \in \Phi_\varphi$ ,  $r \in B$  iff  $\kappa(r) \subseteq B$ . The  $\alpha$ -closure of a set  $B$  is the smallest  $\alpha$ -closed set containing  $B$ .

The following function computes the  $\alpha$ -closure of a set  $B$ :

```
function α-closure (B: set of formulas) returns set of formulas
let S = B
while S is not α-closed do
  if r ∈ Φφ is an α-formula such that r ∈ S but κ(r) ⊈ S
```

```

    then  $S := S \cup \kappa(r)$            --  $\alpha$ -expansion
  if  $r \in \Phi_\varphi$  is an  $\alpha$ -formula such that  $\kappa(r) \subseteq S$  but  $r \notin S$ 
      then  $S := S \cup \{r\}$            --  $\alpha^{-1}$ -expansion
end-while
return  $S$ 

```

**Example** Consider the set  $B$ :  $\{\Box(\Box p \wedge \Box q)\}$ . Applying  $\alpha$ -closure to  $B$  will compute the following sequence of sets:

$$\begin{aligned}
 S_0: & \{ \Box(\Box p \wedge \Box q) \} \\
 S_1: & \{ \Box(\Box p \wedge \Box q), \quad \Box p \wedge \Box q, \quad \Diamond \Box(\Box p \wedge \Box q) \} \\
 S_2: & \{ \Box(\Box p \wedge \Box q), \quad \Box p \wedge \Box q, \quad \Diamond \Box(\Box p \wedge \Box q), \quad \Box p, \quad \Box q \} \\
 S_3: & \{ \Box(\Box p \wedge \Box q), \quad \Box p \wedge \Box q, \quad \Diamond \Box(\Box p \wedge \Box q), \quad \Box p, \quad \Box q, \\
 & \qquad \qquad \qquad p, \quad \Diamond \Box p \} \\
 S_4: & \{ \Box(\Box p \wedge \Box q), \quad \Box p \wedge \Box q, \quad \Diamond \Box(\Box p \wedge \Box q), \quad \Box p, \quad \Box q, \\
 & \qquad \qquad \qquad p, \quad \Diamond \Box p, \quad q, \quad \Diamond \Box q \} = \alpha\text{-closure}(B). \blacksquare
 \end{aligned}$$

Referring to the  $\beta$ -tables of Figures 5.1 and 5.2, we say that a  $\beta$ -formula  $r$  is *supported in* a set of formulas  $B$  if either  $\kappa_1(r) \in B$  or  $\kappa_2(r) \subseteq B$ .

A set  $B$  is said to be  $\beta$ -closed if, for every  $\beta$ -formula  $r \in \Phi_\varphi$ ,  $r \in B$  iff either  $\kappa_1(r) \in B$  or  $\kappa_2(r) \subseteq B$  (or both). A set of formulas  $B$  is called *complete* if, for each  $p \in \Phi_\varphi$ ,  $B$  contains either  $p$  or  $\neg p$ . The set  $B$  is said to be *locally consistent* if it does not contain a formula and its negation and the state formula  $\text{state}(B)$ , i.e., the conjunction of all state formulas in  $B$ , is satisfiable.

**Claim 5.12** (conditions for an atom)

A set of formulas  $B \subseteq \Phi_\varphi$  is an atom iff it is  $\alpha$ -closed,  $\beta$ -closed, locally consistent, and complete.

In **Problem 5.7**, the reader is requested to prove Claim 5.12.

The following recursive function  $\text{cover}(B)$  accepts a set of formulas  $B \subseteq \Phi_\varphi$  and returns as a result the set of all  $\varphi$ -atoms containing  $B$ :

```

recursive function  $\text{cover}(B: \text{set of formulas})$  returns  $\text{set of atoms}$ 
if  $B$  is not locally consistent then return  $\emptyset$            -- no atoms contain  $B$ 
if  $B$  is not  $\alpha$ -closed then return  $\text{cover}(\alpha\text{-closure}(B))$ 
if there exists some  $\beta$ -formula  $r \in \Phi_\varphi$  such that  $r \notin B$  but  $\kappa_1(r) \in B$  or
 $\kappa_2(r) \subseteq B$ 
  then return  $\text{cover}(B \cup \{r\})$            --  $\beta^{-1}$ -expansion

```

```

if there exists some  $\beta$ -formula  $r \in B$  which is not supported in  $B$ 
    then return  $cover(B \cup \{\kappa_1(r)\}) \cup cover(B \cup \kappa_2(r) \cup \{\neg\kappa_1(r)\})$ 
        --  $\beta$ -expansion

if there exists some  $p \in \Phi_\varphi$ , such that  $p \notin B$  and  $\neg p \notin B$ 
    then return  $cover(B \cup \{p\}) \cup cover(B \cup \{\neg p\})$  -- comp-expansion
return  $\{B\}$  --  $B$  is an atom.

```

The function starts by checking if  $B$  is locally consistent. If  $B$  is not locally consistent, there exists no atom that contains  $B$ . In that case,  $cover$  returns the empty set of atoms.

Next, the algorithm checks if  $B$  is  $\alpha$ -closed. If  $B$  is not  $\alpha$ -closed, function  $cover$  is called recursively with an argument which is the  $\alpha$ -closure of  $B$ . If there is a  $\beta$ -formula  $r \in \Phi_\varphi$ , that is not in  $B$  but one of its alternative consequences,  $\kappa_1(r)$  or  $\kappa_2(r)$ , is in  $B$ , function  $cover$  is called recursively with the argument  $B \cup \{r\}$ .

Next, we check whether the other  $R_\beta$  requirements are satisfied. If not, there exists some  $\beta$ -formula  $r$  such that  $r \in B$  but neither  $\kappa_1(r)$  nor  $\kappa_2(r)$  are in  $B$ . At this point, we recursively invoke  $cover$  to compute the cover of  $B \cup \{\kappa_1(r)\}$  and the cover of  $B \cup \kappa_2(r) \cup \{\neg\kappa_1(r)\}$  and return the union of their results. Adding the negation  $\neg\kappa_1(r)$  is not essential to the correctness of the algorithm but it increases its efficiency by ensuring that  $cover(B \cup \{\kappa_1(r)\})$  and  $cover(B \cup \kappa_2(r) \cup \{\neg\kappa_1(r)\})$  return disjoint sets of atoms.

At this point in the algorithm, we know that  $B$  is  $\alpha$ - and  $\beta$ -closed. We therefore proceed to complete the atoms by considering formulas  $p \in \Phi_\varphi$  such that neither  $p$  nor  $\neg p$  belong to  $B$ . Again, we compute recursively the cover of  $B \cup \{p\}$  and the cover of  $B \cup \{\neg p\}$  and return as a result the union of these two covers. If  $B$  is  $\alpha$ - and  $\beta$ -closed and also complete, then  $B$  itself is an atom, and  $cover$  returns the set  $\{B\}$  as the set of atoms covering  $B$ .

## Tree Representation of a Cover Computation

Consider the formula

$$\varphi_4: \diamond p \vee \Box \neg p.$$

In Fig. 5.23 we present a tree that describes the steps taken by function  $cover$  to obtain the set of atoms covering  $\{\varphi_4\}$ . Note that  $\Phi_{\varphi_4}$  is given by  $\Phi_{\varphi_4}^+ = \Phi_{\varphi_4}^+ \cup \Phi_{\varphi_4}^-$ , where

$$\Phi_{\varphi_4}^+: \{\varphi_4, \diamond p, \Box \neg p, p, \Diamond \diamond p, \Box \Box \neg p\}.$$

Each node in the tree is labeled by a set of formulas and represents an intermediate step in the construction of an atom. We denote by  $part(n)$  the partial

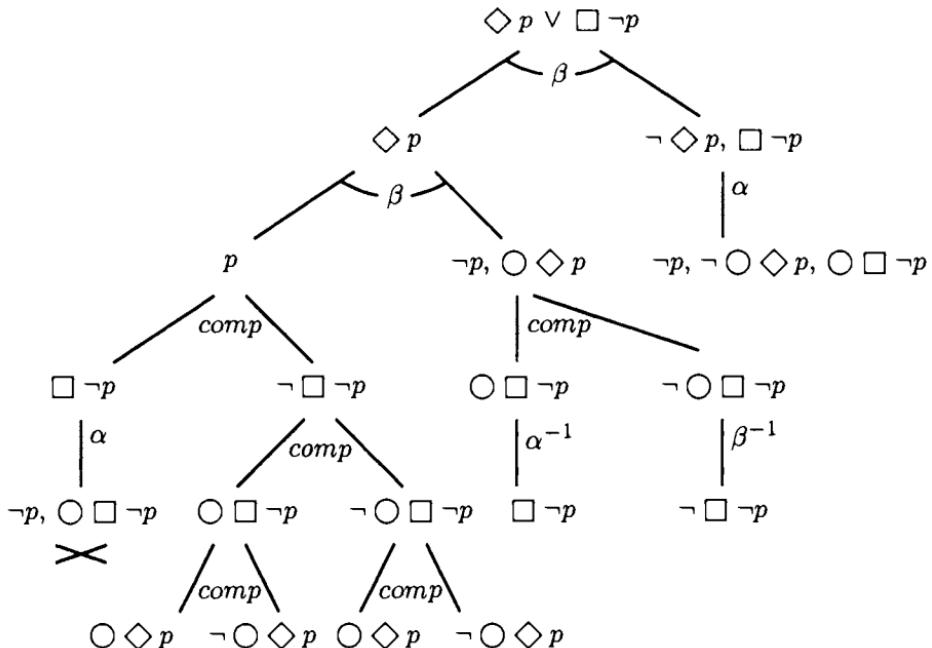


Fig. 5.23. Construction of atoms covering  $\varphi_4$ :  $\diamond p \vee \Box \neg p$ .

atom containing the formulas labeling  $n$  and all of its ancestors. Thus, the node  $n$  labeled by  $p$  represents the partial atom  $\text{part}(n)$ :  $\{\diamond p \vee \Box \neg p, \diamond p, p\}$ . Each nonleaf node is expanded in one of the following ways:

- An  $\alpha$ -expansion may be applied to any node  $n$  such that  $\text{part}(n)$  contains some  $\alpha$ -formula  $r \in \text{part}(n)$ . It creates node  $n'$ , a single child of  $n$ , and labels it with the formulas in  $\kappa(r) - \text{part}(n)$ . For example,  $\alpha$ -expansion is applied to the node labeled by  $\Box \neg p$ , producing a child labeled by  $\neg p, \bigcirc \Box \neg p$ . Similarly,  $\alpha$ -expansion is applied to the node labeled by  $\neg \diamond p, \Box \neg p$ , producing a child labeled by  $\neg p, \neg \bigcirc \diamond p, \bigcirc \Box \neg p$ . Formula  $\neg \bigcirc \diamond p$  is introduced into this child's label as part of  $\kappa(\neg \diamond p)$  from the  $\alpha$ -table of Fig. 5.2 (page 404).
- A  $\beta$ -expansion applied to node  $n$  identifies a  $\beta$ -formula  $r$  that is not supported in  $\text{part}(n)$ . It creates nodes  $n_1$  and  $n_2$ , two children of  $n$ , and labels them by  $\kappa_1(r)$  and  $\{\neg \kappa_1(r)\} \cup \kappa_2(r)$ , respectively. For example,  $\beta$ -expansion is applied to the root resulting in two children labeled by  $\diamond p$  and  $\neg \diamond p, \Box \neg p$ .
- An *inverse  $\alpha$ -expansion ( $\alpha^{-1}$ -expansion)* applied to node  $n$  identifies an  $\alpha$ -formula  $r \in \Phi_\varphi$  such that  $\kappa(r) \subseteq \text{part}(n)$  but  $r \notin \text{part}(n)$ . It creates  $n'$ , a single child of  $n$ , and labels it with  $r$ . For example,  $\alpha^{-1}$ -expansion is applied to the node labeled by  $\bigcirc \Box \neg p$ , yielding the child node labeled by  $\Box \neg p$ .

- An *inverse  $\beta$ -expansion* ( $\beta^{-1}$ -expansion) applied to node  $n$  identifies a  $\beta$ -formula  $r \in \Phi_\varphi$  such that  $\kappa_1(r) \in \text{part}(n)$  or  $\kappa_2(r) \subseteq \text{part}(n)$  but  $r \notin \text{part}(n)$ . It creates  $n'$ , a single child of  $n$ , and labels it with  $r$ . For example,  $\beta^{-1}$ -expansion is applied to the node labeled by  $\neg \bigcirc \square \neg p$ , yielding the child node labeled by  $\neg \square \neg p$ . This expansion uses the  $\beta$ -table entry for  $\neg \square \neg p$ , presented in Fig. 5.2.
- A *completion expansion* (*comp-expansion*) applied to node  $n$  identifies a formula  $r \in \Phi_\varphi$  such that neither  $r$  nor  $\neg r$  belong to  $\text{part}(n)$ . It creates nodes  $n_1$  and  $n_2$ , children of  $n$ , and labels them by  $r$  and  $\neg r$ , respectively.

Whenever an expansion creates a new node  $n$ , we check whether  $\text{part}(n)$  is locally consistent. If  $\text{part}(n)$  is locally inconsistent, we mark it as a *failure* node and no further expansion is applied to it. Graphically, this is represented by an  ~~$\times$~~  mark appearing below the node. For example, the node labeled by  $\{\neg p, \bigcirc \square \neg p\}$  in the tree of Fig. 5.23 is identified as locally inconsistent and marked as failure.

Leaves in the tree are nodes  $n$  that are either locally inconsistent or such that no further expansion can be applied to them. It follows that all the locally consistent leaves of the tree represent atoms.

For example, the leaves of the tree of Fig. 5.23 represent the following atoms covering formula  $\varphi_4$ :  $\Diamond p \vee \square \neg p$ :

$$\begin{aligned}
 A_1: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad p, \quad \neg \square \neg p, \quad \bigcirc \square \neg p, \quad \bigcirc \Diamond p\} \\
 A_2: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad p, \quad \neg \square \neg p, \quad \bigcirc \square \neg p, \quad \neg \bigcirc \Diamond p\} \\
 A_3: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad p, \quad \neg \square \neg p, \quad \neg \bigcirc \square \neg p, \quad \bigcirc \Diamond p\} \\
 A_4: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad p, \quad \neg \square \neg p, \quad \neg \bigcirc \square \neg p, \quad \neg \bigcirc \Diamond p\} \\
 A_5: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad \neg p, \quad \bigcirc \square \neg p, \quad \bigcirc \Diamond p, \quad \square \neg p\} \\
 A_6: & \{\Diamond p \vee \square \neg p, \quad \Diamond p, \quad \neg p, \quad \neg \bigcirc \square \neg p, \quad \bigcirc \Diamond p, \quad \neg \square \neg p\} \\
 A_7: & \{\Diamond p \vee \square \neg p, \quad \neg \Diamond p, \quad \neg p, \quad \neg \bigcirc \Diamond p, \quad \square \neg p, \quad \bigcirc \square \neg p\}
 \end{aligned}$$

The described tree expansion can be taken as a computational alternative to procedure *cover* using the tree data structure to replace recursion.

## Future Candidates

Assume that a certain atom  $A$  has been added to the tableau. A typical step in the incremental tableau construction is the computation of the atoms  $B_1, \dots, B_k$  that can appear as successors of  $A$  in a tableau.

A formula  $\psi \in \Phi_\varphi$  is called an *implied successor* of an atom  $A$  under one of the following cases:

- $\bigcirc \psi \in A$ .
- $\neg \bigcirc \neg \psi \in A$ .
- $\psi = \Theta p$  and  $p \in A$ .
- $\psi = \neg \Theta p$  and  $\neg p \in A$ .
- $\psi = \odot p$  and  $p \in A$ .
- $\psi = \neg \odot p$  and  $\neg p \in A$ .

Obviously, if  $A$  and  $B$  represent the sets of the closure formulas that hold at positions  $j$  and  $j+1$  of a model, respectively, then  $B$  must include all the formulas that are implied successors of  $A$ . For an atom  $A$ , we denote by  $\text{imps}(A)$  the set of all implied successors of  $A$ .

Consider, for example, the formula

$$\varphi: (\bigcirc p \wedge \neg \bigcirc q) \vee (\Theta p \vee \neg \Theta q)$$

and one of its atoms:

$$A: \{ \neg p, \quad q, \quad \bigcirc p, \quad \neg \bigcirc q, \quad \neg \Theta p, \quad \neg \Theta q, \quad \bigcirc p \wedge \neg \bigcirc q, \\ \Theta p \vee \neg \Theta q, \quad \varphi \}$$

The set  $\text{imps}(A)$  is given by

$$\text{imps}(A) = \{ p, \quad \neg q, \quad \neg \Theta p, \quad \Theta q \}$$

The formula  $p$  is included in  $\text{imps}(A)$  because  $\bigcirc p \in A$ . Formula  $\neg q$  is included because  $\neg \bigcirc q \in A$ . Formula  $\neg \Theta p$  is included because  $\neg p \in A$ . Finally, formula  $\Theta q$  is included because  $q \in A$ .

We let

$$\text{successors}(A) = \text{cover}(\text{imps}(A))$$

be the set of all atoms obtained as the cover of the set of implied successors. The set  $\text{successors}(A)$  contains all the atoms  $B_1, \dots, B_k$  that can appear as successors of  $A$  in the tableau.

**Example** The set  $\text{successors}(A)$  of the atom

$$A: \{ \neg p, \quad q, \quad \bigcirc p, \quad \neg \bigcirc q, \quad \neg \Theta p, \quad \neg \Theta q, \quad \bigcirc p \wedge \neg \bigcirc q, \\ \Theta p \vee \neg \Theta q, \quad \varphi \}$$

is given by:

$$B_1: \{ p, \quad \neg q, \quad \neg \Theta p, \quad \Theta q, \quad \neg \bigcirc p, \quad \neg \bigcirc q, \quad \neg(\bigcirc p \wedge \neg \bigcirc q), \\ \neg(\Theta p \vee \neg \Theta q), \quad \neg \varphi \}$$

- $B_2:$   $\{p, \neg q, \neg \Theta p, \Theta q, \neg \bigcirc p, \bigcirc q, \neg(\bigcirc p \wedge \neg \bigcirc q),$   
 $\neg(\Theta p \vee \neg \Theta q), \neg \varphi\}$
- $B_3:$   $\{p, \neg q, \neg \Theta p, \Theta q, \bigcirc p, \neg \bigcirc q, \bigcirc p \wedge \neg \bigcirc q,$   
 $\neg(\Theta p \vee \neg \Theta q), \varphi\}$
- $B_4:$   $\{p, \neg q, \neg \Theta p, \Theta q, \bigcirc p, \bigcirc q, \neg(\bigcirc p \wedge \neg \bigcirc q),$   
 $\neg(\Theta p \vee \neg \Theta q), \neg \varphi\}.$  ■

The following claim states that  $\text{successors}(A)$  correctly computes the set of atoms that are possible successors of  $A$ .

**Claim 5.13** (correctness of  $\text{successors}$ )

Atoms  $A$  and  $B$  satisfy the connection requirements  $R_{\bigcirc}$ ,  $R_{\Theta}$ , and  $R_{\neg \Theta}$  (page 403) iff  $B \in \text{successors}(A)$ .

In Problem 5.8, the reader is requested to prove this claim.

### Incremental Construction of the Tableau

It is now possible to describe an incremental construction of a tableau  $T_\varphi$  for a formula  $\varphi$ .

**Algorithm INCREMENTAL** — incremental tableau construction

- Initially place in  $T_\varphi$  all the atoms contained in  $\text{cover}(\{\varphi\})$ .
- Remove all atoms which are not initial, i.e., atoms that contain a formula of the form  $\Theta p$  or a formula of the form  $\neg \Theta p$ .
- Mark all atoms as unprocessed.
- For each unprocessed atom  $A$ ,
  - let  $S = \text{successors}(A)$ .
  - For each  $B \in S$ ,
    - if  $B$  is not already in  $T_\varphi$ , add  $B$  to  $T_\varphi$  and mark it as unprocessed.
    - draw an edge from  $A$  to  $B$ .
- end-for
- mark  $A$  as processed.
- end-for.

It can be shown that Algorithm INCREMENTAL constructs precisely the part of the tableau which is reachable from initial  $\varphi$ -atoms, without constructing the unreachable atoms.

**Example** Let us apply Algorithm INCREMENTAL to the formula

$$\varphi_1: \square p \wedge \diamond \neg p.$$

The first step of the algorithm requires the computation of  $\text{cover}(\varphi_1)$ . In Fig. 5.24, we present a tree computation of this cover.

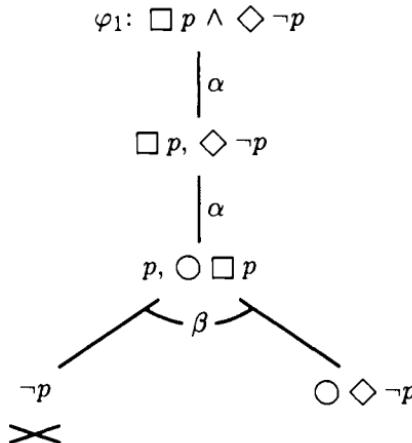


Fig. 5.24. Construction of atoms covering  $\varphi_1: \square p \wedge \diamond \neg p$ .

As we see, the cover consists of a single atom, which is identical to the atom  $A_7$  constructed on page 406:

$$A_7: \{p, \quad \circlearrowleft \square p, \quad \circlearrowleft \diamond \neg p, \quad \square p, \quad \diamond \neg p, \quad \varphi_1\}.$$

Thus, we place atom  $A_7$  in  $T_\varphi$ . Next, we compute  $\text{imps}(A_7)$  which yields:

$$\text{imps}(A_7) = \{\square p, \diamond \neg p\}.$$

Computing the cover of this set, we observe that by  $\alpha^{-1}$ -expansion, any atom in the cover must contain  $\varphi_1$ , which immediately shows that the cover is again given by  $\{A_7\}$ .

This concludes the application of Algorithm INCREMENTAL in this simple case, resulting in a tableau which contains only atom  $A_7$  connected to itself. Indeed, this is precisely the part of the tableau of Fig. 5.3 that is reachable from initial  $\varphi_1$ -atoms. ■

## Additional Simplifications

A significant reduction in the size of the tableau can result from simplifications of temporal formulas. For example, we may apply some of the following simplifications:

$$\begin{array}{lll} \neg\neg p & \longrightarrow & p \\ \square \square p & \longrightarrow & \square p \\ \square p \wedge \square q & \longrightarrow & \square(p \wedge q). \end{array}$$

Another possible optimization is the rewriting of  $\square p$  into  $\neg\lozenge\neg p$  (and possibly for the other temporal operators as well), thus potentially reducing the number of atoms if some of the closure formulas coincide. For example, with this rewriting, the formula

$$\varphi_1: \square p \wedge \lozenge \neg p$$

would become

$$\varphi_1: \neg\lozenge\neg p \wedge \lozenge \neg p.$$

At this point, we may either use boolean simplification (which is allowed within any context) to derive  $F$ , or continue to expand the closure which only contains the following basic formulas:

$$\{p, \bigcirc \lozenge \neg p\},$$

compared to the three basic formulas  $\{p, \bigcirc \square p, \bigcirc \lozenge \neg p\}$ , originally listed in the closure of formula  $\varphi_1$  (page 405).

Another possible optimization could result from the use of a minimal set of temporal operators. For example, all the future operators can be defined in terms of the two operators  $\bigcirc$  and  $\mathcal{U}$ , and all the past operators can be defined in terms of  $\ominus$  and  $\mathcal{S}$ . This could again lead to smaller closures and, consequently, to smaller tableaux.

## 5.5 Particle Tableaux

The tableau considered up to this point consisted of atoms which, by requirement  $R_{\neg}$ , must include  $p$  or  $\neg p$  for every  $p \in \Phi_\varphi$ . This requirement is often too demanding. Consider, for example, the formula  $\varphi: \bigcirc \bigcirc \bigcirc p$ . An atom containing  $\varphi$  represents a position  $j$  in a model which predicts that  $p$  will hold at the position three steps ahead, i.e., at position  $j + 3$ . By  $R_{\neg}$ , the atom representing  $j$  is required to make a commitment about whether  $p$ ,  $\bigcirc p$ , and  $\bigcirc \bigcirc p$  hold or do not hold at position  $j$ . This information is completely irrelevant to the satisfaction of  $\bigcirc \bigcirc \bigcirc p$  at  $j$  and forces us to consider 8 atoms, each interpreting the three formulas  $p$ ,  $\bigcirc p$ , and  $\bigcirc \bigcirc p$ , in a different way.

In this section, we consider approaches which allow us to relax this over-commitment situation, and admit sets of formulas that may commit themselves only with respect to closure formulas that are relevant. This leads to much more efficient tableaux, called *particle tableaux*.

## Incomplete Atoms

As explained above, a significant improvement in the size of generated tableaux can be achieved if we relax the requirement that, for each  $p \in \Phi_\varphi$ , either  $p$  or  $\neg p$  must be included in the atom. We will redevelop here the theory of tableaux, using the relaxed version of atoms.

Without loss of generality, we assume that the formula  $\varphi$  is given in *positive form*, which means that negation is only applied to state formulas. Any formula can be brought to a congruent positive form by repeatedly applying the following rewriting rules to every formula matching their left-hand sides:

$$\begin{array}{ll}
 \neg\neg p & \longrightarrow p \\
 \neg(p \vee q) & \longrightarrow \neg p \wedge \neg q \\
 \neg(p \wedge q) & \longrightarrow \neg p \vee \neg q \\
 \neg\bigcirc p & \longrightarrow \bigcirc \neg p \\
 \neg\lozenge p & \longrightarrow \square \neg p \\
 \neg\square p & \longrightarrow \lozenge \neg p \\
 \neg(p \mathcal{U} q) & \longrightarrow (\neg q) \mathcal{W} (\neg p \wedge \neg q) \\
 \neg(p \mathcal{W} q) & \longrightarrow (\neg q) \mathcal{U} (\neg p \wedge \neg q)
 \end{array}
 \quad
 \begin{array}{ll}
 \neg\bigodot p & \longrightarrow \ominus \neg p \\
 \neg\ominus p & \longrightarrow \bigodot \neg p \\
 \neg\lozenge p & \longrightarrow \boxminus \neg p \\
 \neg\boxminus p & \longrightarrow \lozenge \neg p \\
 \neg(p \mathcal{S} q) & \longrightarrow (\neg q) \mathcal{B} (\neg p \wedge \neg q) \\
 \neg(p \mathcal{B} q) & \longrightarrow (\neg q) \mathcal{S} (\neg p \wedge \neg q).
 \end{array}$$

## Closure and Particles

Since the approach based on incomplete atoms uses formulas in positive form, we restrict our attention to the part of the closure consisting of  $\Phi_\varphi^+$  plus the negation of state formulas.

This portion of the closure, denoted by  $\tilde{\Phi}_\varphi$ , can be independently defined as the smallest set of formulas satisfying the following requirements:

- $\varphi \in \tilde{\Phi}_\varphi$ .
- For every  $p \in \tilde{\Phi}_\varphi$  and  $q$  a subformula of  $p$ ,  $q \in \tilde{\Phi}_\varphi$ .
- For every  $\psi \in \{\square p, \lozenge p, p \mathcal{U} q, p \mathcal{W} q\}$ , if  $\psi \in \tilde{\Phi}_\varphi$  then  $\bigcirc \psi \in \tilde{\Phi}_\varphi$ .
- For every  $\psi \in \{\lozenge p, p \mathcal{S} q\}$ , if  $\psi \in \tilde{\Phi}_\varphi$  then  $\ominus \psi \in \tilde{\Phi}_\varphi$ .
- For every  $\psi \in \{\boxminus p, p \mathcal{B} q\}$ , if  $\psi \in \tilde{\Phi}_\varphi$  then  $\bigodot \psi \in \tilde{\Phi}_\varphi$ .

We define a *particle* (incomplete atom) over  $\varphi$  to be a subset  $A \subseteq \tilde{\Phi}_\varphi$  satisfying the following requirements:

$R_{sat}$ :  $state(A)$  is satisfiable

$I_\neg$ : If  $p \in A$ , then  $\neg p \notin A$

$R_\alpha$ : For every  $\alpha$ -formula  $p \in \tilde{\Phi}_\varphi$ ,  $p \in A$  iff  $\kappa(p) \subseteq A$

$R_\beta$ : For every  $\beta$ -formula  $q \in \tilde{\Phi}_\varphi$ ,  $q \in A$  iff either  $\kappa_1(q) \in A$  or  $\kappa_2(q) \subseteq A$  (or both).

For example, the following are some particles of the formula  $\varphi_4$ :  $\Diamond p \vee \Box \neg p$ :

$$P_1: \{\varphi_4, \Diamond p, p\}$$

$$P_2: \{\varphi_4, \Box \neg p, \neg p, \bigcirc \Box \neg p\}.$$

The conditions for connecting atoms in the tableau are also relaxed. Particle  $A$  may be *connected* to particle  $B$  in a particle tableau whenever the following two *connection requirements* are satisfied:

$I_\bigcirc$ : If  $\bigcirc p \in A$ , then  $p \in B$ .

$I_\ominus$ : If  $\ominus p \in B$  or  $\oslash p \in B$ , then  $p \in A$ .

The connections requirements  $I_\bigcirc$  and  $I_\ominus$  are only *necessary* conditions for connecting particle  $A$  to particle  $B$ . This is different from requirements  $R_\bigcirc$ ,  $R_\ominus$ , and  $R_\oslash$  (page 407) which are necessary *and* sufficient conditions for connecting atom  $A$  to atom  $B$  in an atom tableau. Consequently, conditions  $I_\bigcirc$  and  $I_\ominus$  do not tell us when particles  $A$  and  $B$  *must* be connected. This is determined by the algorithm for constructing a particle tableau which we later present.

Since we no longer have negations of arbitrary formulas, it is also necessary to reformulate the notion of an atom fulfilling a promising formula. A particle  $A$  is said to *fulfill* formula  $\psi$  which promises  $r$  if

$$\psi \notin A \quad \text{or} \quad r \in A.$$

Intending to use particles in an incremental construction, we present the versions of function *cover* and Algorithm INCREMENTAL adapted to deal with particles.

### Cover of a Set of Formulas by Particles

Procedure  $cover_P$  accepts a set of formulas  $B \subseteq \tilde{\Phi}_\varphi$  and returns as result a set of particles which contain  $B$ . We refer to this set of particles as the *particle cover* of  $B$ .

**recursive function**  $cover_P(B: \text{set of formulas})$  **returns set of particles**  
**if**  $B$  is not locally consistent **then return**  $\emptyset$       -- no particles contain  $B$

```

if  $B$  is not  $\alpha$ -closed then return  $\text{cover}_P(\alpha\text{-closure}(B))$ 
if there exists some  $\beta$ -formula  $r \in \tilde{\Phi}_\varphi$  such that  $r \notin B$  but  $\kappa_1(r) \in B$  or
 $\kappa_2(r) \subseteq B$ 
    then return  $\text{cover}_P(B \cup \{r\})$                                 --  $\beta^{-1}$ -expansion
If there exists some  $\beta$ -formula  $r \in B$  which is not supported in  $B$ 
    then return  $\text{cover}_P(B \cup \{\kappa_1(r)\}) \cup \text{cover}_P(B \cup \kappa_2(r))$       --  $\beta$ -expansion
return  $\{B\}$                                               --  $B$  is a particle.

```

**Example** Let us illustrate the application of function  $\text{cover}_P$  to the computation of a set of particles covering the formula set  $\{\varphi_4\}$ , where  $\varphi_4: \diamond p \vee \Box \neg p$ . As before, we can illustrate this computation by a tree. This tree is presented in Fig. 5.25. The particles produced by this cover correspond to the three paths of the tree:

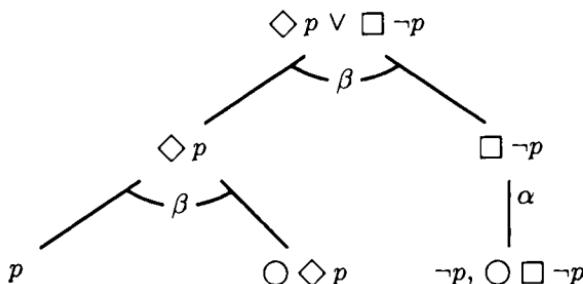


Fig. 5.25. Tree for computing coverage by particles.

- $P_1: \{\diamond p \vee \Box \neg p, \quad \diamond p, \quad p\}$
- $P_2: \{\diamond p \vee \Box \neg p, \quad \diamond p, \quad \bigcirc \diamond p\}$
- $P_3: \{\diamond p \vee \Box \neg p, \quad \Box \neg p, \quad \bigcirc \Box \neg p, \quad \neg p\}$

Compare these three particles to the seven atoms (page 447) produced by  $\text{cover}(\varphi_4)$  (see Fig. 5.23). ■

The previous example showed that  $\text{cover}_P$  produces fewer particles (three) than the number of atoms produced by  $\text{cover}$  (seven) for the same formula. What is less clear is the proper statement of correctness for  $\text{cover}_P$ . One possibility would be to claim that function  $\text{cover}_P$  produces *all* the particles containing  $B$ . This is not true. For example, the particle

$$P_4: \{\Diamond p \vee \Box \neg p, \quad \Diamond p, \quad \bigcirc \Diamond p, \quad p\},$$

which can be obtained as the union of particles  $P_1$  and  $P_2$  is not produced by  $\text{cover}_P$ .

The proper statement of correctness for  $\text{cover}_P$  is that it produces an *adequate cover* for  $B$ . A set of particles  $\{P_1, \dots, P_k\}$  is said to be an adequate cover for a set of formulas  $B$  if, for every atom  $A$  containing  $B$ , there exists a particle  $P_i$ , such that  $B \subseteq P_i \subseteq A$ .

**Claim 5.14** ( $\text{cover}_P$  produces an adequate cover)

The set of particles resulting from the application of the function  $\text{cover}_P$  to a set of formulas  $B \subseteq \tilde{\Phi}_\varphi$  is an adequate cover for  $B$ .

**Example** Let us show that  $K = \{P_1, P_2, P_3\}$  is an adequate cover of the formula set  $B = \{\varphi_4\}$ . In a previous consideration of  $\varphi_4$  (page 446), we derived seven atoms by applying  $\text{cover}$  to  $\varphi_4$ . It is straightforward to check that  $P_1$  covers  $A_1 - A_4$ , that is,  $P_1 \subseteq A_i$  for  $i = 1, \dots, 4$ . In a similar way,  $P_2$  covers  $A_5$  and  $A_6$  and  $P_3$  covers  $A_7$ . ■

In Problem 5.9, the reader is requested to prove Claim 5.14.

### Successors of a Particle

Next, we consider the process of computing the particles that are successors of a given particle  $P$ . Here we face the new problem that not all particles have successors. In fact, there are also atoms that have no successors. For example, atom  $A$ :  $\{\bigcirc p, \bigcirc \neg p, \bigcirc p \wedge \bigcirc \neg p\}$  has no successors, because any successor of  $A$  must contain both  $p$  and  $\neg p$  which is impossible in an atom.

An atom or a particle  $A$  is called *realizable* if there exists a model  $\sigma$  and a position  $j$  such that all formulas in  $A$  hold at position  $j$  of  $\sigma$ . Obviously, only realizable atoms can participate in fulfilling paths. Therefore, it is acceptable that unrealizable atoms, such as  $A$ :  $\{\bigcirc p, \bigcirc \neg p, \bigcirc p \wedge \bigcirc \neg p\}$  do not have successors.

The situation is different with particles. Consider the particle  $P$ :  $\{\bigcirc \ominus p\}$ . This particle is realizable. However, it cannot have a successor  $Q$ . This is because, by the connection requirement  $I_\bigcirc$ , such a successor must contain  $\ominus p$ . By requirement  $I_\ominus$ ,  $\ominus p \in Q$  implies  $p \in P$ , which is not the case. Thus, the problem with particle  $P$  is not that it is not realizable but that it may lack some formulas necessary to maintain the succession relation. This problem arises only in the presence of past operators, which will not be discussed in this chapter. In the rest of the chapter, we will restrict our attention to the case in which  $\varphi$  is a future formula.

## Particle Tableaux for Future Formulas

Assume that  $\varphi$  is a future formula. Then its closure  $\tilde{\Phi}_\varphi$  and all its particles only contain future formulas. We use the definition of an implied successor formula with no change, even though the only relevant case in the definition is that formula  $\psi$  is an implied successor of  $A$  whenever  $\bigcirc \psi \in A$ .

Consequently, we define

$$\text{successors}_P(A) = \text{cover}_P(\text{imps}(A)).$$

With this definition, we can now present an algorithm for the incremental construction of a *particle tableau*  $\tilde{T}_\varphi$  for a future formula  $\varphi$ .

**Algorithm** PART-TAB — particle tableau construction

- Initially place in  $\tilde{T}_\varphi$  all the particles contained in  $\text{cover}_P(\{\varphi\})$ .
- Mark all particles as unprocessed.
- For each unprocessed particle  $P$ ,  
let  $S = \text{successors}_P(P)$ .  
**For each**  $Q \in S$ ,  
if  $Q$  is not already in  $\tilde{T}_\varphi$ , add  $Q$  to  $\tilde{T}_\varphi$  and  
mark it as unprocessed.  
draw an edge from  $P$  to  $Q$   
**end-for**  
mark  $P$  as processed.  
**end-for.**

**Examples** We may now reconsider some of the formulas presented earlier and construct their particle tableaux.

In Fig. 5.26 we present a particle tableau for the formula

$$\varphi_1: \square p \wedge \diamond \neg p$$

which contains a single particle.

$$P_0: \square p \wedge \diamond \neg p, \quad \square p, \quad \diamond \neg p, \quad p, \quad \bigcirc \square p, \quad \bigcirc \diamond \neg p$$

Fig. 5.26. Particle tableau for  $\varphi_1: \square p \wedge \diamond \neg p$ .

For formula  $\varphi_1$ , the particle tableau is identical to the incremental tableau, constructed in page 450, which consisted of the single atom  $A_7$ .

Next, consider  $\neg \varphi_1: \neg(\square p \wedge \diamond \neg p)$ . In positive form, this formula is

$$\psi_1: \diamond \neg p \vee \Box p.$$

In Fig. 5.27, we present the particle tableau for  $\psi_1$ . Note that particle  $P_3$  contains no formulas. This is because when we form the set of implied successors of particle  $P_0$ , we obtain the empty set,  $P_0$  not containing any formula of the form  $\bigcirc q$ . Compare the four particles appearing in this tableau with the eight atoms of Fig. 5.3, reachable from a  $\neg\varphi_1$ -atom. Even though the formula  $\psi_1$  is syntactically different from  $\neg\varphi_1$ , there is a one-to-one correspondence between an incremental tableau for  $\psi_1$  and the incremental tableau for  $\neg\varphi_1$ , both of which consist of eight atoms if we do not apply additional simplifications.

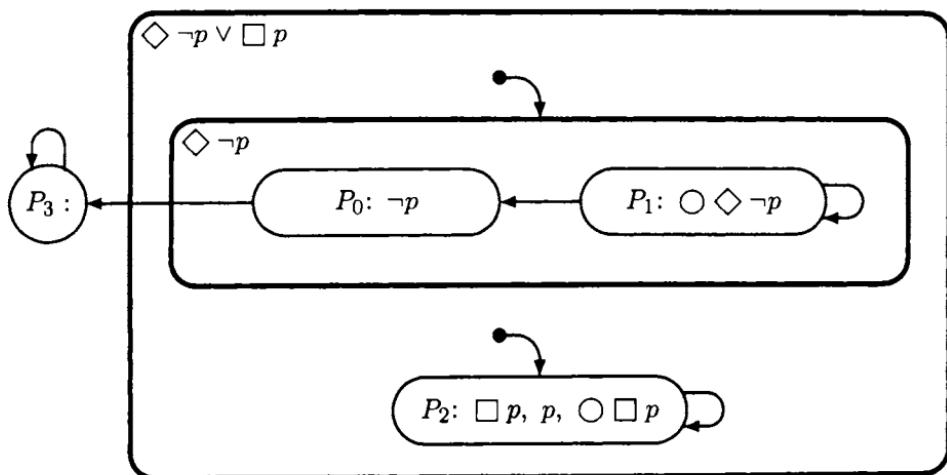


Fig. 5.27. Particle tableau for  $\psi_1: \diamond \neg p \vee \Box p$ .

As our next example, we consider the formula

$$\varphi_2: \Box(\neg at\_l_2 \vee \diamond at\_l_3).$$

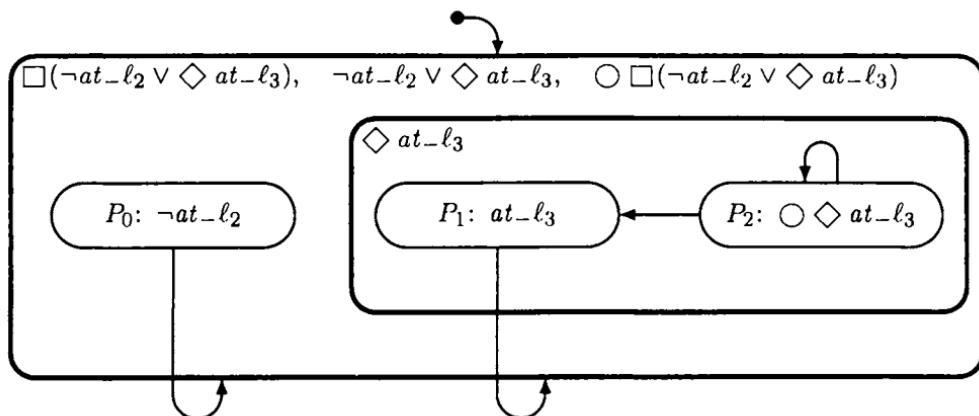
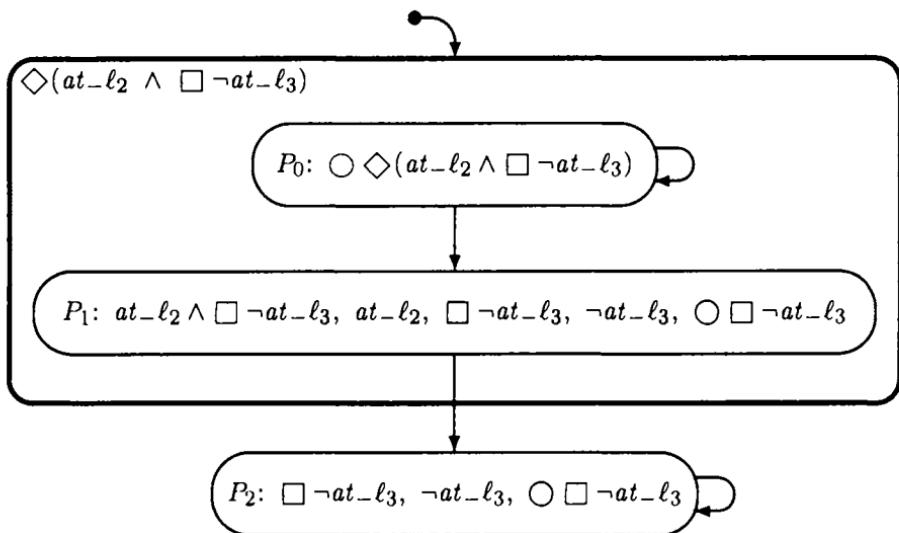
In Fig. 5.28, we present a particle tableau for  $\varphi_2$ , containing three particles. Compare this with the seven-atom tableau, presented in Fig. 5.4.

Finally, we consider  $\neg\varphi_2: \neg \Box(\neg at\_l_2 \vee \diamond at\_l_3)$  given in positive form as

$$\psi_2: \diamond(at\_l_2 \wedge \Box \neg at\_l_3).$$

The particle tableau for  $\psi_2$  is presented in Fig. 5.29. ■

As mentioned earlier, a particle  $Q$  is said to fulfill a formula  $\psi$  which promises  $r$  if either  $\psi \notin Q$  or  $r \in Q$ . Consequently, we may explore particle tableaux for fulfilling MSCS's.

Fig. 5.28. Particle tableau for  $\varphi_2: \Box(\neg at\_l_2 \vee \Diamond at\_l_3)$ .Fig. 5.29. Particle tableau for  $\psi_2: \Diamond(at\_l_2 \wedge \Box \neg at\_l_3)$ .

The correctness of Algorithm PART-TAB is stated by the following claim:

**Claim 5.15** (Algorithm PART-TAB is correct)

A future formula  $\varphi$  is satisfiable iff its particle tableau  $\tilde{T}_\varphi$  contains a  $\varphi$ -

reachable fulfilling MSCSs.

**Justification** A central part in the proof of the claim is to show that, for each model  $\sigma: s_0, s_1, \dots$  satisfying  $\varphi$ , there exists a (fulfilling) path  $\pi: P_0, P_1, \dots$  in  $\tilde{T}_\varphi$  such that  $(\sigma, j) \models p$  for every  $p \in P_j$  and position  $j \geq 0$ . Following the proof of Claim 5.2, we define the sequence of atoms  $A_0, A_1, \dots$  by letting  $A_j$  contain the set of all formulas  $p \in \Phi_\varphi$  such that  $(\sigma, j) \models p$ .

We now construct the sequence  $P_0, P_1, \dots$ , inductively.

- *base case* — Initially, Algorithm PART-TAB placed in  $\tilde{T}_\varphi$  all the particles contained in  $\text{cover}_P(\{\varphi\})$  which, as claimed earlier, is an adequate cover of  $\{\varphi\}$ . Since  $(\sigma, 0) \models \varphi$ , we know that  $\varphi \in A_0$ , and hence, there is some particle  $P_0 \in \tilde{T}_\varphi$  such that  $\varphi \in P_0 \subseteq A_0$ .
- *inductive step* — Assume that we have already constructed the particle sequence  $P_0, P_1, \dots, P_i$  such that  $P_0 \subseteq A_0, \dots, P_i \subseteq A_i$ . We show how to construct  $P_{i+1} \subseteq A_{i+1}$ . Since  $P_i \subseteq A_i$  it is obvious that  $\text{imps}(P_i)$ , the set of formulas required by  $I_\circlearrowleft$  to be included in any successor of  $P_i$ , is contained in  $A_{i+1}$ . As all particles in  $\text{successors}_P(P_i) = \text{cover}_P(\text{imps}(P_i))$  are included in  $\tilde{T}_\varphi$  and form an adequate cover for  $\text{imps}(P_i)$ , one of them, say  $P_{i+1}$ , must be a subset of  $A_{i+1}$ . We thus extend the path  $P_0, P_1, \dots, P_i$  by one more particle  $P_{i+1} \subseteq A_{i+1}$ .

In Problem 5.10, the reader is requested to complete the proof of Claim 5.15. ■

**Examples** Let us review the particle tableaux in Figs. 5.26–5.29 and check which of them corresponds to satisfiable formulas.

The particle tableau of Fig. 5.26 for  $\varphi_1: \Box p \wedge \Diamond \neg p$  contains a single MSCS  $\{P_0\}$  which is not fulfilling. It promises  $\Diamond \neg p$ , which is never fulfilled. We conclude that  $\varphi_1: \Box p \wedge \Diamond \neg p$  is unsatisfiable.

The particle tableau of Fig. 5.27 for  $\psi_1: \Diamond \neg p \vee \Box p$  contains two  $\psi_1$ -reachable fulfilling MSCS's:  $\{P_2\}$  and  $\{P_3\}$ . We conclude that  $\psi_1: \Diamond \neg p \vee \Box p$  is satisfiable. A satisfying model based on  $\{P_3\}$  can be given by

$$\langle p: \mathsf{F} \rangle \langle p: - \rangle^\omega,$$

where the notation  $p: -$  means that the value assigned to  $p$  from the second state on is arbitrary.

A satisfying model based on  $\{P_2\}$  is

$$\langle p: \mathsf{T} \rangle^\omega.$$

The particle tableau of Fig. 5.28 for  $\varphi_2: \Box(\neg \text{at\_}\ell_2 \vee \Diamond \text{at\_}\ell_3)$  contains a single fulfilling MSCS:  $\{P_0, P_1, P_2\}$ . We conclude that  $\varphi_2: \Box(\neg \text{at\_}\ell_2 \vee \Diamond \text{at\_}\ell_3)$  is satisfiable. A possible satisfying model based on this MSCS is given by

$$(\langle \text{at\_}\ell_2: \mathsf{F}, \text{at\_}\ell_3: - \rangle \langle \text{at\_}\ell_2: -, \text{at\_}\ell_3: \mathsf{T} \rangle)^\omega.$$

The only fulfilling MSCS in the particle tableau of Fig. 5.29 for formula  $\psi_2$ :  $\Diamond(at\_l_2 \wedge \Box \neg at\_l_3)$  is  $\{P_2\}$ . We conclude that  $\psi_2$ :  $\Diamond(at\_l_2 \wedge \Box \neg at\_l_3)$  is satisfiable. A satisfying model based on  $\{P_2\}$  is

$$(at\_l_2: T, at\_l_3: F) \langle at\_l_2: -, at\_l_3: F \rangle^\omega.$$

In **Problem 5.11**, the reader is requested to construct particle tableaux for two formulas.

## Problems

**Problem 5.1** ( $\alpha$ - and  $\beta$ -tables for negations) page 404

Construct an  $\alpha$ -table and a  $\beta$ -table for the negations of the formulas:  $p \wedge q$ ,  $p \vee q$ ,  $p \mathcal{U} q$ ,  $p \mathcal{W} q$ ,  $p \mathcal{S} q$ , and  $p \mathcal{B} q$ .

**Problem 5.2** (no fulfillment is necessary for past formulas) page 412

In Claim 5.5 it was stated that only fulfilling paths induce models, where the notion of fulfillment only involved formulas with future principal operators. Here, we ask the reader to show that no corresponding requirement is necessary for past formulas. To show this, please prove the following:

If  $\pi: A_0, A_1, \dots$  is an infinite path in a tableau for  $\varphi$ , where  $A_0$  is an initial atom, then there exists a model  $\sigma$  such that, for every past formula  $\psi \in \Phi_\varphi$  and every  $j \geq 0$

$$(\sigma, j) \models \psi \text{ iff } \psi \in A_j.$$

**Problem 5.3** (lower bound on the size of the tableau) page 417

Present a family of formulas  $\varphi_n$ ,  $n = 1, 2, \dots$ , whose tableaux contain  $\Omega(2^n)$  atoms, and such that the ultimately periodic model for  $\varphi_n$  with the smallest period, has a period of length  $\Omega(2^n)$ .

*Hint:* Consider the propositions  $p_0, \dots, p_{n-1}$ . View any truth assignment to these propositions as the value of a binary counter with  $n$  bits,  $p_0$  being the least significant and  $p_{n-1}$  being the most significant. In any state, interpret  $p_i = F$  to mean that the corresponding bit has the value 0, while  $p_i = T$  encodes the value 1. Write a formula of size  $O(n)$  which specifies a model as follows:

- At position 0,  $p_0, \dots, p_{n-1}$  encode the number 0.
- The value encoded at position  $j + 1$  equals the value encoded at position  $j$  plus 1 modulo  $2^n$ .

**Problem 5.4** (checking satisfiability of example formulas) page 422

Use Algorithm SAT to check satisfiability of the following formulas:

- (a)  $\Diamond \Box p \wedge \neg p$
- (b)  $p \mathcal{U} q \wedge \Box \neg q$
- (c)  $\Box(p \mathcal{U} q \wedge \neg q)$
- (d)  $\Diamond(q \wedge \widehat{\Box} p) \wedge \neg(p \mathcal{U} q)$
- (e)  $\Diamond(\neg \Diamond(q \wedge \widehat{\Box} p) \wedge (p \mathcal{U} q)).$

For formulas that are found to be satisfiable, present a satisfying model, using the  $\omega$ -notation for describing the periodic infinite suffix of the model.

**Problem 5.5** (checking satisfiability over system LOOP<sup>+</sup>) page 434

Use Algorithm P-SAT to check whether the formula

$$\Box \Diamond(x = 1) \wedge \Box(x \neq 3)$$

is satisfiable over system LOOP<sup>+</sup> of Fig. 5.13.

**Problem 5.6** (additional properties of Program MUX-PET1) page 443

Use the methods of Section 5.3 to check whether the following formulas are valid over Program MUX-PET1 of Fig. 5.19

- (a)  $at\_l_3 \wedge at\_m_2 \Rightarrow (\neg at\_m_4) \mathcal{U} (at\_l_4)$
- (b)  $at\_l_4 \wedge at\_m_3 \Rightarrow (at\_l_{3,4}) \mathcal{S} (\neg at\_m_3).$

In your construction, you may restrict your attention to atoms in which at most one of  $\{at\_l_3, at\_l_4\}$  and at most one of  $\{at\_m_2, at\_m_3, at\_m_4\}$  is true.

**Problem 5.7** (conditions for being an atom) page 444

Prove Claim 5.12, by which a set of formulas  $B \subseteq \Phi_\varphi$  is a  $\varphi$ -atom iff  $B$  is  $\alpha$ -closed,  $\beta$ -closed, locally consistent, and complete.

**Problem 5.8** (correctness of successors) page 449

Prove Claim 5.13 which states that  $\text{successors}(A) = \text{cover}(\text{imps}(A))$  contains the set of all atoms that can appear as successors of  $A$  in a tableau.

**Problem 5.9** (correctness of Claim 5.14) page 455

Prove Claim 5.14 which states that the set of particles resulting from the application of the function  $\text{cover}_P$  to a set of formulas  $B \subseteq \tilde{\Phi}_\varphi$  is an adequate cover for  $B$ .

**Problem 5.10** (correctness of Claim 5.15) page 459

Complete the proof of Claim 5.15 which states that a future formula  $\varphi$  is satisfiable iff the particle tableau  $T_\varphi$ , constructed by Algorithm PART-TAB, contains a  $\varphi$ -reachable fulfilling MSCS.

**Problem 5.11** (constructing particle tableaux) page 460

Use Algorithm PART-TAB for an incremental construction of a particle tableau for the formulas (b)–(c) presented in Problem 5.4.

## Bibliographic Remarks

**The tableau:** The main approach to the algorithms presented in this chapter is based on the construction of a tableau for a temporal formula. The use of semantic tableaux in first-order logic as a basis for a semi-decision procedure and for proving completeness of the predicate calculus is discussed in Smullyan [1968]. Semantic tableaux for modal logics (of which temporal logic is a special case) are presented in Hughes and Cresswell [1968]. Fischer and Ladner [1979] present a decision procedure for propositional dynamic logic (PDL), based on the observation that if a PDL formula has a model it has a model whose size is at most exponential in the size of the formula. Such a model can always be presented as a structure containing worlds (corresponding to our atoms), in which each world is uniquely identified by the interpretation it assigns to a set of formulas that are defined as the *closure* of the original formula. Thus, the notion of closure used in this chapter is taken from Fischer and Ladner [1979] and is often called “the Fischer-Ladner closure” of the formula. However, Fischer and Ladner did not propose a systematic construction of this structure (tableau).

Such a systematic construction was proposed by Pratt [1980] who suggested a tableau construction very similar to the one presented in Section 5.1. Ben-Ari, Manna, and Pnueli [1983] and Clarke and Emerson [1981] present a tableau-based decision algorithm for the branching-time logic CTL (called UB in Ben-Ari, Manna, and Pnueli [1983]). It is straightforward to derive from a branching-time decision procedure an algorithm for checking satisfiability of linear-time formulas. Such an algorithm is presented in Wolper [1983] and Manna and Wolper [1984]. Wolper [1983] also points out some limitations of the standard temporal logic with suggestions for their removal.

**Complexity:** Sistla and Clarke [1985] established the complexity bounds for checking satisfiability of (linear-time) temporal formulas and for the problem of model checking, which we refer to in this book as the algorithmic verification problem. They showed that both these problems are PSPACE complete which implies that, at the current state of knowledge, the best we can hope for is an

exponential algorithm. In comparison, Clarke and Emerson [1981] showed that satisfiability checking for branching-time logic is exponential-time complete while model checking is polynomial.

A more concrete model-checking algorithm, based on forming the Cartesian product of the system with a tableau for the considered formula, which is the method described in Section 5.2, is presented in Lichtenstein and Pnueli [1984]. The paper also refines the complexity analysis by showing that the presented algorithm is polynomial in the size of the program and exponential only in the size of the formula to be verified. This analysis can easily be extended to show that the model-checking problem for linear-time temporal logic is polynomial in the size of the program and  $\text{PSPACE}$  complete in the size of the formula. This model-checking algorithm was extended to the full temporal language, including the past operators, in Lichtenstein, Pnueli, and Zuck [1985].

**Automata techniques:** Tableaux are only one of the possible approaches to checking satisfiability and performing model checking of linear-time temporal formulas. Another widely considered approach is based on  $\omega$ -automata, and on the translation of a temporal formula into an  $\omega$ -automaton that specifies the same set of infinite state sequences (models). A good reference for the theory of automata on infinite objects, including  $\omega$ -automata, is presented by Thomas [1990]. The use of  $\omega$ -automata to specify behaviors of systems was suggested first in Muller [1963]. Particular applications of the method with attention to efficient implementation are discussed by Aggarwal, Kurshan, and Sabnani [1983].

The translation of temporal formulas into  $\omega$ -automata and using this translation for checking satisfiability was described in Manna and Wolper [1981] and in Wolper, Vardi, and Sistla [1983], whose final versions are presented in Manna and Wolper [1984] and Vardi and Wolper [1994], respectively. Automata-theoretic approaches to model checking are based on translating the negation of the formula into an automaton and checking for emptiness of the intersection of this automaton with an automaton representing the program. This approach was developed in Vardi and Wolper [1986]. The book by Kurshan [1994] presents a comprehensive methodology for verification of finite-state systems, using automata-theoretic methods. Efficient algorithms for checking emptiness of such automata are presented by Emerson [1985]. Additional references to the relations between temporal logics, tableaux, and automata are available in Emerson [1990] and Wolper [1989].

**Efficient tableau construction:** Incremental tableau construction for first-order logic and temporal and modal logics is described in Smullyan [1968], Rescher and Urquhart [1971], and Hughes and Cresswell [1968]. The construction of the branching-time tableaux of Ben-Ari, Manna, and Pnueli [1983] is also incremental. The incremental construction described in Section 5.5 is based on Kesten, Manna, McGuire, and Pnueli [1993]. The algorithm described in that paper is more general than the one presented here and also applies to formulas with past operators. This algorithm was implemented in STeP, the Stanford Temporal

Prover, as reported in Manna et al. [1994].

**Symbolic model checking:** The approach to the verification of finite-state systems known as *symbolic model checking* is described in Burch et al. [1992] and in Burch et al. [1994]. Ordered Binary Decision Diagrams (OBDDs), introduced by Bryant [1986, 1992], are a compact representation for boolean functions and relations, and can reduce the *state explosion problem* that arises from explicitly representing the state spaces of large systems. The SMV system of McMillan [1993] is an OBDD-based model checker for branching-time logic. Clarke, Grumberg and Hamaguchi [1994] show that it can be adapted to perform linear-time model checking as well. See the book by Kurshan [1994] for more references.

## References

- Abadi, M. [1989]. The power of temporal proofs. *Theor. Comp. Sci.* 65:35–84.
- Abadi, M. and L. Lamport [1988]. The existence of refinement mappings. *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, 165–177.
- Abadi, M. and L. Lamport [1993]. Composing specifications. *ACM Trans. on Prog. Lang. and Sys.*, 15:73–132.
- Abadi, M. and Z. Manna [1990]. Nonclausal deduction in first-order temporal logic. *J. ACM*, 37(2):279–317.
- Aggarwal, S., R. P. Kurshan, and K. Sabnani [1983]. A calculus for protocol specification and validation. In *Protocol Specification, Testing and Verification III*, North-Holland, Amsterdam, 19–34.
- Alpern, B. and F.B. Schneider [1985]. Defining liveness. *Inf. Proc. Letters*, 21(4):181–185.
- Alpern, B. and F.B. Schneider [1987]. Recognizing safety and liveness. *Dist. Comp.*, 2:117–126.
- Andrews, G.R. [1991]. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Redwood City, CA.
- Apt, K.R. [1981]. Ten years of Hoare's logic: A survey – part I. *ACM Trans. on Prog. Lang. and Sys.*, 3:431–483.
- Apt, K.R., N. Francez, and S. Katz [1988]. Appraising fairness in languages for distributed programming. *Dist. Comp.*, 2:226–241.
- Apt, K.R., N. Francez, and W.-P. de Roever [1980]. A proof system for communicating sequential processes. *ACM Trans. on Prog. Lang. and Sys.*, 2:359–385.
- Apt, K.R. and E.R. Olderog [1991]. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, Berlin.

- Arnold, A. [1983]. Topological characterizations of infinite behaviors of transition systems. *Proc. 10th Int. Colloq. Lang. Prog.* Lec. Notes in Comp. Sci. 154, Springer-Verlag, Berlin, 28–38.
- Arnold, A. [1985]. A syntactic congruence for rational  $\omega$ -languages. *Theor. Comp. Sci.* 39:333–335.
- Ashcroft, E.A. [1975]. Proving assertions about parallel programs. *J. Comp. Sys. Sci.*, 10:110–135.
- Ashcroft, E.A. and Z. Manna [1971]. Formalization of properties of parallel programs. In *Machine Intelligence 6*, Edinburgh Univ. Press, 17–41.
- Back, R.-J.R. [1988]. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624.
- de Bakker, J.W. [1980]. *Mathematical Theory of Program Correctness*. Prentice-Hall, Englewood Cliffs, NJ.
- de Bakker, J.W., W.-P. de Roever, and G. Rozenberg (editors) [1990]. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. Lec. Notes in Comp. Sci. 430, Springer-Verlag, Berlin.
- Bacon, J. [1980]. Substance and first-order quantification over individual concepts. *J. Symb. Logic*, 45:193–203.
- Barringer, H. [1985]. *A Survey of Verification Techniques for Parallel Programs*. Lec. Notes in Comp. Sci. 191, Springer-Verlag, Berlin.
- Barringer, H., R. Kuiper, and A. Pnueli [1984]. Now you may compose temporal logic specifications. *Proc. 16th ACM Symp. on Theory of Comp.*, 51–63.
- Barringer, H., R. Kuiper, and A. Pnueli [1985]. A compositional temporal approach to a CSP-like language. In *Formal Models of Programming*, E.J. Neuhold and G. Chroust (editors), IFIP, North-Holland, Amsterdam, 207–227.
- Ben-Ari, M. [1984]. Algorithms for on-the-fly garbage collection. *ACM Trans. on Prog. Lang. and Sys.*, 6(3):333–344.
- Ben-Ari, M. [1990]. *Principles of Concurrent Programming*. Prentice-Hall, London.
- Ben-Ari, M., Z. Manna, and A. Pnueli [1983]. The temporal logic of branching time. *Acta Informatica*, 20(3):207–26.
- van Benthem, J.F.A.K. [1983]. *The Logic of Time*. Reidel, Dordrecht.
- Bochmann, G.V. [1982]. Hardware specification with temporal logic: An example. *IEEE Trans. on Computers*, C-31(3).
- de Bruijn, N.G. [1967]. Additional comments on a problem in concurrent programming control. *Comm. ACM*, 8(9):137–138.

- Bryant, R.E. [1986]. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691.
- Bryant, R.E. [1992]. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.
- Büchi, J.R. [1960]. Weak second order arithmetic and finite automata. *Zeitschrift für Math. Log. und Gründl. der Math.*, 6:66–92.
- Burch, J.R., E.M. Clarke, D.E. Long, K.L. McMillan, and D. L. Dill [1994]. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424.
- Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang [1992]. Symbolic ModelChecking:  $10^{20}$  States and Beyond. *Inf. and Comp.*, 98(2):142–170.
- Burgess, J.P. [1990]. Basic tense logic. In *Handbook of Philosophical Logic*, Vol. II, D. Gabbay and F. Guenther (editors), Reidel, Dordrecht, 89–133.
- Burstall, R.M. [1974]. Program proving as hand simulation with a little induction. *Proc. IFIP Information Processing 74*, North-Holland, Amsterdam, 308–312.
- Caplain, M. [1975]. Finding invariant assertions for proving programs. *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA.
- Chandy, K.M. and J. Misra [1984]. The drinking philosophers problem. *ACM Trans. on Prog. Lang. and Sys.*, 6(4):632–646.
- Chandy, K.M. and J. Misra [1988]. *Parallel Program Design*. Addison-Wesley, Reading, MA.
- Chang, E. [1994]. *Compositional Verification of Reactive and Real-time Systems*. Ph.D. Thesis, Computer Science Dept., Stanford Univ., Stanford, CA.
- Chang, E., Z. Manna, and A. Pnueli [1994]. Compositional verification of real-time systems. In *Proc. 9th IEEE Symp. Logic in Comp. Sci.*, 458–465.
- Chellas, B.F. [1980]. *Modal Logic: An Introduction*, Cambridge Univ. Press, Cambridge.
- Clarke, E.M. [1979]. Synthesis of resource invariants for concurrent programs. *Proc. 6th ACM Symp. on Princ. of Prog. Lang.*, 211–221.
- Clarke, E.M. and E.A. Emerson [1981]. Design and synthesis of synchronization skeletons using branching time temporal logic. *Proc. IBM Workshop on Logics of Programs*. Lec. Notes in Comp. Sci. 131, Springer-Verlag, Berlin, 52–71.
- Clarke, E.M., O. Grumberg, and K. Hamaguchi [1994]. Another look at LTL model checking. In *Proc. 6th Conference on Computer Aided Verification*, D.L. Dill (editor). Lec. Notes in Comp. Sci. 818, Springer-Verlag, Berlin, 415–427.

- Collier, W.W. [1968]. System deadlocks. IBM Tech. Report TR-001756.
- Cook, S.A. [1978]. Soundness and completeness of an axiom system for program verification. *SIAM J. Comp.*, 7:70–90.
- Cook, S.A., C. Dwork, and R. Reischuk [1986]. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comp.*, 15(1):87–97.
- Cormen, T.H., C.E. Leiserson, and R.L. Rivest [1990]. *Introduction to Algorithms*. MIT Press, Cambridge, MA.
- Courtois, P.J., F. Heymans, and D.L. Parnas [1971]. Concurrent control with “readers” and “writers.” *Comm. ACM*, 14(10):667–668.
- Cousot, P. and N. Halbwachs [1978]. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symp. Princ. of Prog. Lang.*, 84–96.
- Dershowitz N. and Z. Manna [1981]. Inference rules for program annotation. *IEEE Trans. on Software Eng.*, SE-7(2):207–222.
- Diaz, M. [1982]. Modelling and analysis of communication and cooperation protocols using Petri net based models. Tutorial paper in *Second International Workshop on Protocol Specification, Testing and Verification*, Idyllwild, Los Angeles.
- Dijkstra, E.W. [1965]. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569.
- Dijkstra, E.W. [1968a]. Cooperating sequential processes. In *Programming Languages*, F. Genuys (editor), Academic Press, New York, 43–112.
- Dijkstra, E.W. [1968b]. The structure of the “THE” multiprogramming system. *Comm. ACM* 11(5):341–346.
- Dijkstra, E.W. [1971]. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138.
- Dijkstra, E.W. [1975]. Guarded commands, nondeterminacy, and formal derivation of programs. *Comm. ACM*, 18(8):453–457.
- Dijkstra, E.W. [1976]. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Dijkstra, E.W., W.H. Feijen, and A.J.M. van Gasteren [1983]. Derivation of a termination detection algorithm for distributed computations. *Inf. Proc. Letters*, 16(5):217–219.
- Dijkstra, E.W., L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens [1978]. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, 21(11):966–975.

- Dijkstra, E.W. and C.S. Scholten [1980]. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1-4.
- Dill, D.L., A.J. Drexler, A.J. Hu, and C.H. Yang [1992]. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society Press, 522-525.
- Elspas, B. [1974]. The semiautomatic generation of inductive assertions for proving program correctness. Research Report, SRI, Menlo Park, CA.
- Emerson, E.A. [1985]. Automata, tableaux, and temporal logic. *Proc. Conf. Logics of Programs*, R. Parikh (editor). Lec. Notes in Comp. Sci. 193, Springer-Verlag, Berlin, 79-88.
- Emerson, E.A. [1990]. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen (editor), North-Holland, Amsterdam, 997-1072.
- Emerson, E.A. and E.M. Clarke [1982]. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comp. Prog.*, 2:241-266.
- Enderton, H.B. [1972]. *A Mathematical Introduction to Logic*. Academic Press, New York.
- Fischer, M.J. and R.E. Ladner [1979]. Propositional dynamic logic of regular programs. *J. Comp. Sys. Sci.*, 18:194-211.
- Flon, L. and N. Suzuki [1978]. Consistent and complete proof rules for the total correctness of parallel programs. *Proc. 19th IEEE Symp. on Found. of Comp. Sci.*
- Floyd, R.W. [1967]. Assigning meanings to programs. In *Proc. Symp. on Appl. Math.*, American Mathematical Society, 19:19-32.
- Francez, N. [1976]. *The Analysis of Cyclic Programs*. Ph.D. Thesis, Weizmann Institute, Rehovot.
- Francez, N. [1980]. Distributed termination. *ACM Trans. on Prog. Lang. and Sys.*, 2(1):42-55.
- Francez, N. [1986]. *Fairness*. Springer-Verlag, New York.
- Francez, N. [1992]. *Program Verification*. Addison-Wesley, Reading, MA.
- Francez, N. and A. Pnueli [1978]. A proof method for cyclic programs. *Acta Informatica*, 9:133-157.
- Francez, N., M. Rodeh, and M. Sintzoff [1981]. Distributed termination with interval assertions. *Proc. Int. Colloq. on Formalization of Programming Concepts (EATCS)*, J. Diaz (editor), Penniscola, Spain. Lec. Notes in Comp. Sci. 107, Springer-Verlag, Berlin.

- Gabbay, D. [1976]. *Investigations in Modal and Tense Logics with Applications to Problems in Philosophy and Linguistics*. Reidel, Dordrecht.
- Gabbay, D., A. Pnueli, S. Shelah, and J. Stavi [1980]. On the temporal analysis of fairness. *Proc. 7th ACM Symp. on Princ. of Prog. Lang.*, 163–173.
- Garson, J.W. [1984]. Quantification in modal logic. In *Handbook of Philosophical Logic*, Vol. II, D. Gabbay and F. Guenthner (editors), Reidel, Dordrecht, 249–307.
- German, S.M. [1974]. *A Program Verifier that Generates Inductive Assertions*. Undergraduate Thesis, Harvard Univ., Cambridge, MA.
- Goldblatt, R. [1987]. Logics of time and computation. CSLI Lecture Notes 7, Center for Study of Language and Information, Stanford Univ., Stanford, CA.
- Goldstine, H.H. and J. von Neumann [1947]. Planning and coding problems for an electronic computer. In *Collected Works of John von Neumann* Vol. 5, A.H. Traub (editor), Pergamon Press, New York (1963), 80–235.
- Granger, P. [1991]. Static analysis of linear congruence equalities among variables of a program. In *Proc. of International Joint Conf. on Theory and Practice of Software Development*, S. Abramsky and T.S.E. Maibaum (editors). Lec. Notes in Comp. Sci. 493, Springer-Verlag, Berlin, 169–192.
- Gries, D. [1981]. *The Science of Programming*. Springer-Verlag, New York.
- Habermann, A.N. [1969]. Prevention of system deadlocks. *Comm. ACM*, 12(7): 373–377, 385.
- Hailpern, B.T. [1982]. *Verifying Concurrent Processes Using Temporal Logic*. Lec. Notes in Comp. Sci. 129., Springer-Verlag, Berlin.
- Hailpern, B.T. and S.S. Owicki [1983]. Modular verification of computer communication protocols. *IEEE Trans. on Comm.*, COM-31, 1:56–68.
- Harel, D. [1979]. *First-Order Dynamic Logic*. Lec. Notes in Comp. Sci. 68, Springer-Verlag, Berlin.
- Harel, D. [1987]. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274.
- Havender, J.W. [1968]. Avoiding deadlock in multi-tasking systems. *IBM Syst. J.*, 2:7.
- He, J. [1989]. Process simulation and refinement. *Formal Aspects of Computing Sci.* 1(3):229–241.
- Hoare, C.A.R. [1969]. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580.

- Hoare, C.A.R. [1978]. Communicating sequential processes. *Comm. ACM*, 21(8): 666–677.
- Hoare, C.A.R. [1984]. *Communicating Sequential Processes*. Prentice-Hall, London.
- Hoare, C.A.R., J. He, and J.W. Sanders [1987]. Prespecification in data refinement. *Information Processing Letters*, 25:71–76.
- Holt, R.C., G.S. Graham, E.D. Lazowska, and M.A. Scott [1978]. *Structured Concurrent Programming with Operating System Applications*. Addison-Wesley, Reading, MA.
- Holzmann, G.J. [1991]. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ.
- Hoogeboom, H.J. and G. Rozenberg [1986]. Infinitary languages: Basic theory and applications to concurrent systems. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 224, Springer-Verlag, Berlin, 266–342.
- Hooman, J. and W.-P. de Roever [1986]. The quest goes on: A survey of proof systems for partial correctness of CSP. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 224, Springer-Verlag, Berlin, 343–395.
- Hughes, G.E. and M.J. Cresswell [1968]. *An Introduction to Modal Logic*. Methuen, New York.
- Jones, C.B. [1986]. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, NJ.
- Jonsson, B. [1987]. Modular verification of asynchronous networks. *Proc. 6th ACM Symp. on Princ. of Dist. Comp.*, 152–166.
- Jonsson, B. [1990]. On decomposing and refining specifications of distributed systems. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 430, Springer-Verlag, Berlin, 361–387.
- Kahn, G. [1974]. The semantics of a simple language for parallel programming. *Proc. IFIP Congress 74*, North-Holland, Amsterdam, 471–475.
- Kaminski, M. [1985]. A classification of  $\omega$ -regular languages. *Theor. Comp. Sci.*, 36:217–220.
- Kamp, J.A.W. [1968]. *Tense Logic and the Theory of Linear Order*. Ph.D. Thesis, Michigan State Univ.
- Karr, M. [1976]. Affine relationship among variables of a program. *Acta Informatica*, 6:133–151.

- Katz, S. and Z. Manna [1973]. A heuristic approach to program verification. *Proc. 3rd Int. Joint Conf. on Artificial Intelligence*, Stanford, CA, 500–512.
- Katz, S. and Z. Manna [1976]. Logical analysis of programs. *Comm. ACM*, 19(4): 188–206.
- Keller, R.M. [1976]. Formal verification of parallel programs. *Comm. ACM*, 19(7):371–384.
- Kesten, Y., Z. Manna, H. McGuire, and A. Pnueli [1993]. A decision algorithm for full propositional temporal logic. In *5th Conference on Computer Aided Verification*, C. Courcoubetis (editor), Lec. Notes in Comp. Sci. 697, Springer-Verlag, Berlin, 97–109.
- Knuth, D.E. [1966]. Additional comments on a problem in concurrent program control. *Comm. ACM*, 9(5):321.
- Koymans, R. [1987]. Specifying message buffers requires extending temporal logic. *Proc. 6th ACM Symp. on Princ. of Dist. Comp.*, 191–204.
- Koymans, R. and W.-P. de Roever [1983]. Examples of a real-time temporal logic specification. In *The Analysis of Concurrent Systems*, M.I. Jackson and M.J. Wray (editors). Lec. Notes in Comp. Sci. 207, Springer-Verlag, Berlin, 231–252.
- Kripke, S.A. [1963]. Semantical analysis of modal logic I: normal propositional calculi. *Z. Math. Grund. Math.* 9:67–96.
- Kröger, F. [1977]. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8:243–246.
- Kurshan, R.P. [1994]. *Computer Aided Verification of Coordinating Processes*. Princeton University Press.
- Kyng, M. [1983]. Specification and verification of networks in a Petri net based language. In *Application and Theory of Petri nets*, A. Pagnoni and G. Rozenberg (editors). Informatik Fachberichte 66, Springer-Verlag.
- Ladner, R.E. [1977]. Applications of model-theoretic games to discrete linear orders and finite automata. *Info. and Cont.* 33:281–303.
- Lamport, L. [1974]. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455.
- Lamport, L. [1976]. The synchronization of independent processes. *Acta Informatica*, 7(1):15–34.
- Lamport, L. [1977]. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Eng.*, SE-3(2):125–143.
- Lamport, L. [1983a]. Specifying concurrent program modules. *ACM Trans. on Prog. Lang. and Sys.*, 5(2):190–222.

- Lamport, L. [1983b]. What good is temporal logic? *Proc. IFIP 9th World Congress*, R.E.A. Mason (editor), North-Holland, 657–668.
- Lamport, L. [1985a]. Basic concepts. In *Distributed Systems — Methods and Tools for Specification*. Lec. Notes in Comp. Sci. 190, Springer-Verlag, Berlin, 19–30.
- Lamport, L. [1985b]. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. *Proc. 12th ACM Symp. on Princ. of Prog. Lang.*, New Orleans, LA, 79–83.
- Lamport, L. [1994]. The temporal logic of actions. *ACM Trans. Prog. Lang. Sys.*, 16(3):872–923.
- Landweber, L.H. [1969]. Decision problems for  $\omega$ -automata. *Math. Sys. Theory*, 4:376–384.
- Lautenbach, K. [1972]. Liveness in Petri nets. Internal Report GMD-ISF 72–02.1.
- Levin, G.M. and D. Gries [1981]. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302.
- Lichtenstein, O. and A. Pnueli [1984]. Checking that finite-state concurrent programs satisfy their linear specification. *Proc. 11th ACM Symp. Princ. of Prog. Lang.*, 97–107.
- Lichtenstein, O., A. Pnueli, and L. Zuck [1985]. The glory of the past. *Proc. Conf. on Logics of Programs*. Lec. Notes in Comp. Sci. 193, Springer-Verlag, Berlin, 196–218.
- Lynch, N. [1990]. Multivalued possibilities mappings. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 430, Springer-Verlag, Berlin, 519–543.
- Lynch, N.A. and M.R. Tuttle [1987]. Hierarchical correctness proofs for distributed algorithms. *Proc. 6th Symp. on Princ. of Dist. Comp.*, 137–151.
- Malachi, Y. and S.S. Owicki [1981]. Temporal specifications of self-timed systems. In *VLSI Systems and Computations*, H.T. Kung, B. Sproull, and G. Steels (editors), Computer Science Press.
- Manna, Z. [1969]. Properties of programs and the first-order predicate calculus. *J. ACM*, 16(2):244–255.
- Manna, Z. [1974]. *Mathematical Theory of Computation*. McGraw-Hill, New York.
- Manna, Z. [1982]. Verification of sequential programs: Temporal axiomatization. In *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt (editors), NATO Advanced Study Institutes Series, D. Reidel, Dordrecht, 53–102.

- Manna, Z., A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T. Uribe [1994]. STeP: The Stanford Temporal Prover. Computer Science report, Stanford Univ., Stanford, CA.
- Manna, Z. and A. Pnueli [1979]. The modal logic of programs. *Proc. 6th Int. Colloq. Aut. Lang. Prog.* Lec. Notes in Comp. Sci. 71, Springer-Verlag, Berlin, 385–409.
- Manna, Z. and A. Pnueli [1981a]. Verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science*, R.S. Boyer and J.S. Moore (editors), Academic Press, London, 215–273.
- Manna, Z. and A. Pnueli [1981b]. Verification of concurrent programs: Temporal proof principles. In *Proc. Workshop on Logics of Programs*, D.C. Kozen (editor), Lec. Notes in Comp. Sci. 131, Springer-Verlag, Berlin, 200–252.
- Manna, Z. and A. Pnueli [1983a]. How to cook a temporal proof system for your pet language. *Proc. 10th ACM Symp. on Princ. of Prog. Lang.*, 141–154.
- Manna, Z. and A. Pnueli [1983b]. Proving precedence properties: The temporal way. *Proc. 10th Int. Colloq. Aut. Lang. Prog.* Lec. Notes in Comp. Sci. 154, Springer-Verlag, Berlin, 491–512.
- Manna, Z. and A. Pnueli [1983c]. Verification of concurrent programs: A temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2*, J.W. de Bakker and J. van Leeuwen (editors), Mathematical Centre Tracts 159, Center for Mathematics and Computer Science, Amsterdam, 163–255.
- Manna, Z. and A. Pnueli [1990]. A hierarchy of temporal properties. *Proc. 9th ACM Symp. on Princ. of Dist. Comp.*, 377–408.
- Manna, Z. and A. Pnueli [1991a]. Completing the temporal picture. *Theor. Comp. Sci.*, 83(1):97–130.
- Manna, Z. and A. Pnueli [1991b]. Tools and rules for the practicing verifier. In *CMU Computer Science: A 25-year Commemorative*, R.F. Rashid (editor), ACM Press and Addison-Wesley, Reading, MA, 125–159.
- Manna, Z. and A. Pnueli [1994]. Temporal verification diagrams. In *Theoretical Aspects of Computer Software*, T. Ito and A.R. Meyer, editors, Lec. Notes in Comp. Sci. 789, Springer-Verlag, Berlin, 726–765.
- Manna, Z. and R. Waldinger [1978]. Is ‘sometime’ sometimes better than ‘always’?: Intermittent assertions in proving program correctness. *Comm. ACM*, 21(2):159–172.
- Manna, Z. and P. Wolper [1981]. Synthesis of communicating processes from temporal-logic specifications. *Proc. IBM Workshop on Logics of Programs*. Lec. Notes in Comp. Sci. 131, Springer-Verlag, Berlin, 253–281.

- Manna, Z. and P. Wolper [1984]. Synthesis of communicating processes from temporal-logic specifications. *ACM Trans. on Prog. Lang. and Sys.*, 6(1):68-93.
- Martinez, J. and M. Silva [1982]. A simple and fast algorithm to obtain all invariants of a generalized Petri net. In *Application and Theory of Petri Nets*, C. Girault and W. Reisig (editors), Informatik Fachbericht 52, Springer Publishing Company.
- Mattern, F. [1987]. Algorithms for distributed termination detection. *Dist. Comp.*, 2:161-175.
- McMillan, K.L. [1993]. *Symbolic Model Checking*. Kluwer Academic Publishers.
- McNaughton, R. and S. Papert [1971]. *Counter-Free Automata*. MIT Press, Cambridge, MA.
- McTaggart, J.M.E. [1927]. *The Nature of Existence*, Vol. I, Cambridge.
- Merritt, M. [1990]. Completeness theorems for automata. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 430, Springer-Verlag, Berlin, 544-560.
- Milner, R. [1980]. *A Calculus of Communicating Systems*. Lec. Notes in Comp. Sci. 92, Springer-Verlag, Berlin.
- Milner, R. [1989]. *Communication and Concurrency*. Prentice-Hall, London.
- Misra, J. [1975]. Relations uniformly conserved by a loop. *Proc. Int. Symp. on Proving and Improving Programs*, Arc-et-Senans, France.
- Misra, J. [1983]. Detecting termination of distributed computations using markers. *Proc. 2nd ACM Symp. on Princ. of Dist. Comp.*, 290-294.
- Misra, J. and K.M. Chandy [1982]. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. on Prog. Lang. and Sys.*, 4(1):37-43.
- Misra, J., K.M. Chandy, and T. Smith [1982]. Proving safety and liveness of communicating processes with examples. *Proc. ACM Symp. on Princ. of Dist. Comp.*, Ottawa, Canada, 157-164.
- Morgan, C. [1990]. *Programming from Specifications*. Prentice-Hall, London.
- Moriconi, M.S. [1974]. Towards the interactive synthesis of assertions. Research Report, Univ. of Texas at Austin.
- Morris, J.M. [1987]. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):287-306.
- Muller, D.E. [1963]. Infinite sequences and finite machines. *Proc. 4th IEEE Symp. Switching Circuit Theory and Logical Design*, 3-16.

- Naur, P. [1966]. Proof of algorithms by general snapshots. *BIT*, 6, 310–316.
- Nguyen, V., A. Demers, S. Owicki, and D. Gries [1986]. A modal and temporal proof system for networks of processes. *Dist. Comp.*, 1(1):7–25.
- Nguyen, V., D. Gries, and S. Owicki [1985]. A model and temporal proof system for networks of processes. *Proc. 12th ACM Symp. on Princ. of Prog. Lang.*, 121–131.
- Owicki, S. and D. Gries [1976a]. An axiomatic proof technique for parallel programs, *Acta Informatica*, 6(4):319–340.
- Owicki, S. and D. Gries [1976b]. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM*, 19(5):279–284.
- Owicki, S. and L. Lamport [1982]. Proving liveness properties of concurrent programs. *ACM Trans. on Prog. Lang. and Sys.*, 4(3):455–495.
- Park, D. [1980]. On the semantics of fair parallelism. In *Abstract Software Specification*. Lec. Notes in Comp. Sci. 86, Springer-Verlag, Berlin, 504–524.
- Park, D. [1981]. A predicate transformer for weak fair interaction. *Proc. 6th IBM Symp. on Mathematical Foundations of Computer Science*, Hakone, Japan.
- Park, D. [1983]. The fairness problem and nondeterministic computing networks. In *Foundations of Computer Science IV, Distributed Systems*, J.W. de Bakker and J. van Leeuwen (editors), Mathematical Centre Tracts 159, Center for Mathematics and Computer Science, Amsterdam, 133–161.
- Perrin, D. [1984]. Recent results on automata and infinite words. *Mathematical Foundations of Comp. Sci.* Lec. Notes in Comp. Sci. 176, Springer-Verlag, Berlin, 134–148.
- Perrin, D. [1985]. Variétés de semigroupes et mots infinis. *C.R. Acad. Sci. Paris*, 295:595–598.
- Perrin, D. and J.E. Pin [1986]. First order logic and star-free sets. *J. Comp. Syst. Sci.* 32:393–406.
- Peterson, G.L. [1983]. A new solution to Lamport's concurrent programming problem. *ACM Trans. on Prog. Lang. and Sys.*, 5(1):56–65.
- Peterson, J.L. [1981]. *Petri Net Theory and Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ.
- Peterson, J.L. and A. Silberschatz [1985]. *Operating Systems Concepts*. Addison-Wesley, Reading, MA.
- Pnueli, A. [1977]. The temporal logic of programs. *Proc. 18th IEEE Symp. on Found. of Comp. Sci.*, 46–57.
- Pnueli, A. [1981]. The temporal semantics of concurrent programs. *Theor. Comp. Sci.*, 13:1–20.

- Pnueli, A. [1985]. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K.R. Apt (editor), sub-series F: Computer and System Science, Springer-Verlag, Berlin, 123–144.
- Pnueli, A. [1986]. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 224, Springer-Verlag, Berlin, 510–584.
- Pratt, V.R. [1980]. A near-optimal method for reasoning about action. *J. Comp. and Sys. Sci.*, 20(2):231–254.
- Prior, A. [1967]. *Past, Present, and Future*. Clarendon Press, Oxford.
- Quelle, J.P. and J. Sifakis [1982]. Specification and verification of concurrent systems in Cesar. In *Proc. Int. Symp. Programming*, M. Dezani-Ciancaglini and M. Montanari (editors). Lec. Notes in Comp. Sci. 137, Springer-Verlag, Berlin, 337–351.
- Raynal, M [1986]. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, MA.
- Reisig, W. [1985]. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer-Verlag, Berlin.
- Rescher, N. and A. Urquhart [1971]. *Temporal Logic*. Springer-Verlag, New York.
- de Roever, W.-P. [1985]. The quest for compositionality — A survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables. In *The Role of Abstract Models in Computer Science*, E.J. Neuhold (editor), North-Holland, Amsterdam, 181–206.
- Schneider, F.B. and G. Andrews [1986]. Concepts for concurrent programming. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lec. Notes in Comp. Sci. 224, Springer-Verlag, Berlin, 669–716.
- Sistla, A.P. [1983]. *Theoretical Issues in the Design of Distributed and Concurrent Systems*. Ph.D. Thesis, Harvard Univ., Cambridge, MA.
- Sistla, A.P. [1985]. On characterization of safety and liveness properties in temporal logic. *Proc. 4th ACM Symp. on Princ. of Dist. Comp.*, 39–48.
- Sistla, A.P. and E.M. Clarke [1985]. The complexity of propositional linear temporal logic. *J. ACM*, 32:733–749.
- Sistla, A.P., E.M. Clarke, N. Francez, and Y. Gurevish [1982]. Can message buffers be characterized in linear temporal logic? *Proc. ACM Symp. on Princ. of Dist. Comp.*, 148–156.

- Sistla, A.P., E.M. Clarke, N. Francez, and A.R. Meyer [1984]. Can message buffers be axiomatized in temporal logic? *Info. and Cont.*, 63(1,2):88–112.
- Sistla, A.P., M.Y. Vardi, and P. Wolper [1987]. The complementation problem for Büchi automata with applications to temporal logic. *Theor. Comp. Sci.*, 49:217–237.
- Smullyan, R. [1968]. *First-order Logic*. Springer-Verlag, Berlin.
- Smullyan, R. [1982a]. *Chess Mysteries of Sherlock Holmes: 50 Tantalizing Problems of Chess detection*. Oxford University Press.
- Smullyan, R. [1982b]. *Chess Mysteries of The Arabian Knights: 50 New Problems of Chess detection*. Oxford University Press.
- Staiger, L. [1987]. Research in the theory of  $\omega$ -languages. *J. Inform. Process. Cybernet.* 23:415–439.
- Stark, E.W. [1988]. Proving entailment between conceptual state specifications. *Theor. Comp. Sci.* 56:135–154.
- Stirling, C. [1992]. Modal and temporal logics. In *Handbook of Logic in Computer Science*, Vol. 2, S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum (editors), Clarendon Press, Oxford, 477–563.
- Stomp, F.A., W.-P. de Roever, and R.T. Gerth [1989]. The  $\mu$ -calculus as an assertion language for fairness arguments. *Inf. and Comp.*, 82:278–322.
- Tarjan, R. [1972]. Depth-first search and linear graph algorithms. *SIAM J. Comp.*, 1:146–160.
- Thomas, W. [1979]. Star-free regular sets of  $\omega$ -sequences. *Info. and Cont.*, 42:148–156.
- Thomas, W. [1981]. A combinatorial approach to the theory of  $\omega$ -automata. *Info. and Cont.*, 48:261–283.
- Thomas, W. [1990]. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, Vol. B, J. van Leeuwen (editor), North-Holland, Amsterdam, 134–191.
- Turing, A. [1950]. Checking a large routine. In *Proc. Conf. on High Speed Automatic Calculating Machines*, McLennan Laboratory, University of Toronto.
- Valiant, L.G. [1975]. Parallelism comparison problems. *SIAM J. Comp.*, 4(3):348–355.
- Vardi, M.Y. and P. Wolper [1986]. An automata-theoretic approach to automatic program verification. *Proc. 1st IEEE Symp. on Logic in Comp. Sci.*, 332–344.
- Vardi, M.Y. and P. Wolper [1994]. Reasoning about infinite computations. *Inf. and Comp.*, 115(1):1–37.

- Wagner, K. [1979]. On  $\omega$ -regular sets. *Info. and Cont.*, 43:123–177.
- Wegbreit, B. [1974]. The synthesis of loop predicates. *Comm. ACM*, 17(2):102–112.
- Wolper, P. [1983]. Temporal logic can be more expressive. *Info. and Cont.*, 56:72–99.
- Wolper, P. [1986]. Expressing interesting properties of programs in propositional temporal logic. *Proc. 13th ACM Symp. on Princ. of Prog. Lang.*, 184–193, St. Petersburg Beach, FL.
- Wolper, P. [1989]. On the relation of programs and computations to models of temporal logic. *Temporal Logic in Specification*, B. Banerjee, H. Barringer, and A. Pnueli (editors). Lec. Notes in Comp. Sci. 398, Springer-Verlag, Berlin, 75–123.
- Wolper, P., M.Y. Vardi, and A.P. Sistla [1983]. Reasoning about infinite computation paths. *Proc. 24th IEEE Symp. on Found. of Comp. Sci.*, 185–194.
- Zwiers, J. [1989]. *Concurrency and Partial Correctness – Proof Theories for Networks of Processes and their Relationship*. Lec. Notes in Comp. Sci. 321, Springer-Verlag, Berlin.



# *Index to Symbols*

## ENGLISH ALPHABET

$\mathcal{A}_{(P,\psi)}$ ,  $\mathcal{A}$ : reachable-atoms graph, 384.

$a_\sigma$ :  $a_0, a_1, \dots$ : atom sequence induced by model  $\sigma$ , 387.

$\mathcal{B}_{(P,\varphi)}$ : behavior graph for finite-state program  $P$  and formula  $\varphi$ , 424.

$C$ : cycle in program, 207.

$C$ : set of compassionate transitions, 3.

$C_\tau$ : enabling condition of transition  $\tau$ , 86.

$\mathcal{E}(y)$ : expression, 102.

$G = (N, E)$ : directed graph, 298.

$G_P$ : state-transition graph for finite-state program  $P$ , 228, 300.

$G_P^-$ : pruned state-transition graph, 300.

$h_p, h_c$ : history variables for producer-consumer, 357.

$F$ : false, 5, 7.

$I_\bigcirc, I_\ominus$ : connection requirements for particle tableau, 453.

$J$ : set of just transitions, 3.

$K$ : control assertion, 113.

$K$ : right constant of linear invariant, 202.

$[\ell]$ : location, 14.

$\ell$ : pre-label, 9.

$\widehat{\ell}$ : post-label, 9.

$\ell^a$ : a label equivalent to  $\ell$  or  $\ell^b$ , 14.

$\ell[j]$ : location in parameterized program, 170.

$L_i$ : processes currently residing at location  $\ell_i$ , 174.

$L_{i_1, i_2, \dots, i_k}, L_{i..j}$ : extension of  $L_i$  to location sets, 175.

$L_M$ : set of locations in module  $M$ , 38.

$\mathcal{L}$ : assertion language, 42.

$\mathcal{L}$ : set of locations in program, 202.

$\mathcal{L}_i$ : set of locations in top-level process  $P_i$ , 109.

$\overline{m}$ : set of all transitions different from  $m$ , 274.

$N_i$ : number of processes currently residing at location  $\ell_i$ , 174.

$N_{i_1, i_2, \dots, i_k}, N_{i..j}$ : extension of  $N_i$  to location sets, 175.

$\mathbb{N}$ : the natural numbers, 225.

$P[1], \dots, P[M]$ : processes, 169.

$R_\bigcirc, R_\ominus, R_\Theta$ : connection requirements, 407.

$R_{sat}, I_-, R_\alpha, R_\beta$ : requirements on particles, 453.

$R_{sat}$ ,  $R_{\neg}$ ,  $R_\alpha$ ,  $R_\beta$ : requirements on atoms, 403.

$\mathcal{R}(s)$ : minimal rank of state  $s$ , 297.

$s$ : state, 2.

$s'$ : a successor state of  $s$ , 3.

$\hat{s}$ : variant of state  $s$ , 4.

$s[u]$ : interpretation of  $u$  by  $s$ , 2.

$\top$ : true, 5, 7.

$T_\varphi$ : tableau, 407.

$\tilde{T}_\varphi$ : particle tableau, 456.

$V$ : system variables, 2.

$V'$ : primed version of system variables, 3.

$\mathcal{V}$ : vocabulary, 2.

$x'$ : primed version of variable  $x$ , 3.

$y^0$ : initial value of variable  $y$ , 203.

$\mathbb{Z}$ : the integers, 222.

## FUNCTIONS AND RELATIONS

$[\alpha \geq], [\alpha \leq], [\alpha \leq \geq]$ : communication events (no values), 57.

$[\alpha \geq v]$ : receiving event, 55.

$[\alpha \leq v]$ : sending event, 55.

$[\alpha \leq \geq v]$ : sending-receiving event, 57.

$\alpha * \beta$ : concatenation of lists, 334, 360.

$\alpha \bullet v$ : append, 20, 55, 358.

$\sigma_1 \cdot \sigma_2$ : concatenation of computations, 302.

$\sim_L$ : label equivalence relation, 13.

$h_c \sqsubseteq h_p$ : prefix inclusion, 361, 362.

$i \ominus_M 1$ : decrement modulo  $M$ , 195.

$j \oplus_M 1$ : increment modulo  $M$ , 172.

## GREEK ALPHABET

$\Gamma_i$ : steps in backward propagation, 261.

$\Gamma_f$ : final step in backward propagation, 261.

$\delta$ : data assertion, 113.

$\delta(A)$ : set of successors of atom  $A$ , 424.

$\delta(i, j)$ : cyclic distance, 284.

$\delta[S]$ : data relation for grouped statements, 25.

$\Delta(V)$ : backward propagation, 260.

$\Delta(e, C)$ : increment of expression  $e$  along cycle  $C$ , 207.

$\Delta(y, \tau)$ : increment to variable  $y$  by transition  $\tau$ , 203.

$\vartheta$ : trail, 424.

$\vartheta_\pi$ : atom sequence induced by path  $\pi$ , 426.

$\Theta$ : initial condition, 2.

$\kappa$ : ranges over classes of properties, 59.

$\kappa$ : test in a program, 105.

$\kappa(\varphi)$  (set of formulas): column in  $\alpha$ -table, 402, 404.

$\kappa_1(\psi)$  (formula): column in  $\beta$ -table, 402, 404.

$\kappa_2(\psi)$  (set of formulas): column in  $\beta$ -table, 402, 404.

$\Lambda$ : empty channel, list 12, 19, 38.

$\pi$ : the control variable, 18.

$\pi$ : initialized path, 426.

$\pi_\ell$ : set of  $\ell_i$  locations contained in  $\pi$ , 94.

$\pi_\sigma$ : atom path induced by model  $\sigma$ , 409.

$\Pi$ : property, 61.

- $\rho_\ell[j]$ : parameterized transition relation, 170.
- $\rho_{(\ell,m)}$ : transition relation for synchronous communication, 21.
- $\rho_\ell^E, \rho_\ell^X$ : transition relations for entry and exit transitions, of cooperation statement, 24.
- $\rho_\ell^T, \rho_\ell^F$ : transition relation disjuncts corresponding to T, F branches, 23.
- $\rho_T(V, V')$ : transition relation, 3.
- $\rho_T^T, \rho_T^F$ : modes of transition relation  $\rho_T$ , 82.
- $\rho_I$ : transition relation of idling transition, 3.
- $\sigma$ :  $s_0, s_1, \dots$ : computation, 3.
- $\sigma$ :  $s_0, s_1, \dots$ : model, 43.
- $\tilde{\sigma}$ : predictive version of computation  $\sigma$ , 84, 324.
- $\hat{\sigma}$ :  $U$ -variant of state sequence  $\sigma$ , 4.
- $\sigma_\pi$ : state sequence induced by path  $\pi$ , 426.
- $\Sigma$ : set of all states, 2.
- $\tau$ : transition, 2.
- $\tau(s)$ :  $\tau$ -successor of state  $s$ , 2.
- $\tau_\ell^E, \tau_\ell^X$ : entry and exit transitions, for cooperation statement, 24.
- $\tau^T, \tau^F$ : modes of transition  $\tau$ , 23, 82.
- $\tau_{(\ell,m)}$ : synchronous communication transition, 21, 56.
- $\tau_E$ : environment transition, 38.
- $\tau_I$ : idling transition, 3.
- $T$ : set of transitions, 2.
- $T^-$ : set of diligent transitions, 3.
- $\varphi^*$ : virtualized version of assertion  $\varphi$ , 137.
- $(\varphi)_0$ : initial version of past formula  $\varphi$ , 318, 320.
- $\Phi_i$ : steps in forward propagation, 258.
- $\Phi_t$ : final step in forward propagation, 259.
- $\Phi_\varphi$  (set of formulas): closure of formula  $\varphi$ , 401.
- $\Phi_\varphi^+$  (set of formulas): positive closure formulas, 402.
- $\Phi_\varphi^-$  (set of formulas): negative closure formulas, 402.
- $\tilde{\Phi}_\varphi$ : closure for particle tableau, 452.
- $\chi$ : previously established  $P$ -invariant, 97.
- $\psi'$ : primed version of assertion  $\psi$ , 82.
- $\psi'$ : primed version of past formula  $\psi$ , 318, 321.
- $\Psi(V)$ : forward propagation, 258.

## NAMES

- $acc_P(\bar{y})$ : accessibility assertion, 221, 223, 224.
- $at'_\ell$ : control predicate,  $[\ell] \in \pi'$ , 18.
- $at_\ell$ : control predicate ( $[\ell] \in \pi$ ), 18.
- $at_{-\ell_{i,j}}, at_{-\ell_{i..j}}$ : abbreviated disjunctions of  $at_\ell$ 's, 55.
- $at_{-L}$ : control predicate,  $L \subseteq \pi$ , 103.
- $cons(u)$ : consume value  $u$ , 313, 331.
- $comm(\ell, m, v)$ : synchronous communication between  $\ell$  and  $m$ , 56.
- $cover(B)$ : atoms covering the formula set  $B$ , 444.
- $cover_P(B)$ : particles covering the formula set  $B$ , 456.
- $div$ : integer-division operator, 137.

- $env(u)$ : possible modification of variable  $u$  by environment, 38.
- $evolve$ : subformula of  $acc_P$  in completeness proof, 223, 225.
- $En(\tau)$ : enabledness of transition  $\tau$ , 3.
- $even(x)$ :  $x$  is an even integer, 90, 222.
- $first$ : formula characterizing first position in model, 46.
- $gcd(m, n)$ : greatest common divisor of  $m$  and  $n$ , 153.
- $hd(\alpha)$ : head (first element) of list  $\alpha$ , 335.
- $imps(A)$ : set of all implied successors of atom  $A$ , 448.
- $in\_S$ : control resides in  $S$ , 55.
- $init$ : subformula of  $acc_P$  in completeness proof, 223, 225.
- $Inf(\pi)$ : all atoms that appear infinitely many times in path  $\pi$ , 427.
- $last$ : subformula of  $acc_P$  in completeness proof, 223, 225.
- $last(\alpha)$ : last element of list  $\alpha$ , 335.
- $lcm(m, n)$ : least common multiple of  $m$  and  $n$ , 154.
- $lowest(N, t)$ : smallest index satisfying a condition, 300.
- $maximal(z, x)$ :  $z$  is the maximal element of the array  $x[1..M]$ , 177.
- $mod$ : modulo, 153.
- $move(L, \hat{L})$ : simultaneous movement of control from locations  $L$  to locations  $\hat{L}$ , 19, 86.
- $move(\ell, \hat{\ell})$ : movement of control from location  $\ell$  to  $\hat{\ell}$ , 19.
- $odd(y)$ :  $y$  is an odd integer, 291.
- $part(n)$ : partial atom for node  $n$ , 445.
- $post(S)$ : post-location of statement  $S$ , 15.
- $post(\tau, \Phi)$ : postcondition of  $\Phi$  via  $\tau$ , 258.
- $pre(\tau, \varphi)$ : precondition of assertion  $\varphi$  with respect to transition  $\tau$ , 114, 260.
- $precede(u_1, u_2, send^*)$ :  $u_1$  precedes  $u_2$  in the pipeline  $send^*$ , 355.
- $pres(U)$ : values of variables in  $U$  are preserved, 19.
- $prime(u)$ :  $u$  is a prime number, 58.
- $prod(u)$ : produce value  $u$ , 313, 331.
- $send^\#$ : virtual version of channel  $send$ , 360.
- $send^*$ : virtual version of channel  $send$ , 334.
- $send^a$ : auxiliary variable corresponding to channel  $send$ , 335.
- $stat(q)$ : statification of  $q$ , 373.
- $stat^{-1}(\chi)$ : unstatified version of assertion  $\chi$ , 377.
- $state(A)$ : conjunction of all state formulas in atom  $A$ , 424.
- $successors(A)$ : set of possible successors of atom  $A$ , 448.
- $successors_P(A)$ : set of possible successors of particle  $A$ , 456.
- $tl(\alpha)$ : tail (list minus first element) of list  $\alpha$ , 335.
- $update(x, k, e)$ : array  $x$  updated at subscript  $k$  with value  $e$ , 171.

## NOTATION

 $(\dots)^\omega$ :  $\omega$ -notation, 400.

$(s, A)$ : node (state  $s$  and atom  $A$ ), 424.

$(s', A')$ : successor node of  $(s, A)$ , 427.

$[s_a, \dots, s_b]$ : segment of states, 288.

$\alpha$ :  $[p_1 \mapsto \varphi_1, \dots, p_r \mapsto \varphi_r]$ : temporal instantiation, 326.

$\Gamma[\bar{y} \mapsto \bar{e}]$ : replacing each occurrence of  $y_i$  in  $\Gamma$  by the corresponding  $e_i$ , 260.

$|T_\varphi|$ : size of tableau, 415.

$(N, E)$ : graph with nodes  $N$  and edges  $E$ , 298.

$\{\varphi\} T \{\psi\}, \{\varphi\} \tau \{\psi\}$ : verification conditions of  $\varphi$  and  $\psi$ , relative to  $T, \tau$ , 84, 319.

$\{\varphi\} \tau \{\psi\}$ : verification condition (proof obligation) of  $\varphi$  and  $\psi$ , relative to transition  $\tau$ , 82, 318.

$\{\varphi\} \tau^{-1} \{\psi\}$ : inverse verification condition, 342.

$\langle V, \Theta, T, \mathcal{J}, \mathcal{C} \rangle$ : fair transition system, 2.

$\emptyset$ : empty set, 2.

$\ell \sim \tilde{\ell}$ : transition from  $\ell$  to  $\tilde{\ell}$ , 103.

$\langle \widehat{V}, \widehat{\Theta}, \widehat{T}, \widehat{\mathcal{J}}, \widehat{\mathcal{C}} \rangle$ : augmented fair transition system, 144.

$\psi[\bar{y} \mapsto \bar{y}']$ : replacing each occurrence of  $y_i$  in  $\psi$  by  $y'_i$ , 86.

$\sigma[a..b], \sigma[a..\infty]$ : segments of computation  $\sigma$ , 288.

$\tau$  while  $at\_L$ : special verification condition, 103.

$at\_l + at\_m$ : arithmetization of boolean values, 94.

$p :=$ : arbitrary truth assignment to  $p$ , 459.

$P_1, \dots, P_k \vdash C$ : proof rule, 54.

: compound node (in verification diagram), 279.

: failure node in tree representation of cover, 447.

: edge between nodes (in verification diagrams), 272.

: terminal node (in verification diagrams), 272.

## PROGRAMMING LANGUAGE

**array**: structured type, 9.

**auxiliary**: mode, 147.

**await**  $c$ : await statement, 5, 20.

**boolean**: basic type, 9.

**channel**: declaration of synchronous channel, 12.

**channel** [1..]: declaration of asynchronous channel, 12.

**character**: basic type, 9.

**consume**  $y$ : consume statement, 7, 23.

**critical**: critical statement, 6, 22.

**for**  $i := 1$  to  $m$  do  $S$ : for statement, 8.

**halt**: halt statement, 5.

**idle**: idle statement, 6, 22.

**if**  $c$  then  $S_1$ : one-branch conditional statement, 7, 23.

**if**  $c$  then  $S_1$  else  $S_2$ : conditional statement, 7, 23.

**if**  $c_1 \rightarrow S_1 \sqcap \dots \sqcap c_k \rightarrow S_k$   
**fi**: conditional selection of guarded commands, 8.

**in**: mode of declared variables, 9,

11.

- integer:** basic type, 9.
- $\ell: [\ell_1: S_1; \widehat{\ell}_1:] \parallel \cdots \parallel [\ell_k: S_k; \widehat{\ell}_k:]$ ;  $\widehat{\ell}::$  cooperation statement, 9, 24.
- list:** structured type, 9.
- local:** mode of declared variables, 9, 11.
- loop forever do**  $S$ : loop-forever statement, 8, 207.
- module:** a module header, 37.
- natural:** basic type, 163.
- noncritical:** noncritical statement, 6, 22.
- out:** mode of declared variables, 9, 11.
- own in, own out:** modes in module declaration, 37.
- prefer**  $S_1$  **to**  $S_2$ : prefer-to statement, 183.
- produce**  $x$ : produce statement, 6, 22.
- $P :: [\text{declaration}; [P_1 :: [\ell_1: S_1; \widehat{\ell}_1:] \parallel \cdots \parallel P_k :: [\ell_k: S_k; \widehat{\ell}_k:]]]$ : program, 11, 62.
- release**  $r$ : release statement, 6, 22.
- release**  $(y, c)$ : generalized release statement, 189.
- request**  $r$ : request statement, 6, 21.
- request**  $(y, c)$ : generalized request statement, 189.
- set:** structured type, 9.
- skip:** skip statement, 5, 20.
- testandset**( $x$ ): test-and-set statement, 161.
- when**  $c$  **do**  $S$ : when statement, 8.
- where**  $\varphi_i$ : constraint on initial value, 9, 11.
- while**  $c$  **do**  $S$ : while statement, 8, 24.
- WHILE**  $c$  **DO**  $S$ : irreproducible while statement, 66.
- $x ::= y$ : swap statement, 160.
- $S_1; \dots; S_k$ : concatenation statement, 7.
- $S_1 \text{ or } \dots \text{ or } S_k$ : selection statement, 8.
- [local declaration;  $S$ ]: block, 9.
- $[\ell_i: S_i; \widehat{\ell}_i:]$ : process in cooperation statement, 9.
- $\alpha \Leftarrow e$ : send statement, 5, 20, 21.
- $\alpha \Rightarrow u$ : receive statement, 6, 21.
- $\langle S \rangle$ : grouped statement, 10, 25.
- $\bigvee_{j=1}^M S[j]$ : parameterized selection statement, 169.
- $\bigvee_{j=1}^M S[j]$ : parameterized cooperation statement, 169.
- $\overline{u} := \overline{e}$ : multiple assignment statement, 5, 20.
- TEMPORAL LOGIC
- $\neg$ : negation, 42, 43.
- $\wedge$ : conjunction, 43.
- $\vee$ : disjunction, 42, 43.
- $\rightarrow$ : implication, 43.
- $\leftrightarrow$ : equivalence, 43.
- $\exists$ : there-exists quantifier, 42, 45.
- $\forall$ : for-all quantifier, 42, 45.
- $\Box p$ : henceforth  $p$ , 43, 44.
- $\Diamond p$ : eventually  $p$ , 43, 44.
- $\bigcirc p$ : next  $p$ , 43, 44.
- $\square p$ : so-far  $p$ , 43, 44.
- $\lozenge p$ : once  $p$ , 43, 44.

- $\odot p$ : before  $p$ , 43, 45.  
 $\ominus p$ : previously  $p$ , 43, 45, 49.  
 $p \mathcal{U} q$ :  $p$  until  $q$ , 43, 44.  
 $p \mathcal{W} q$ :  $p$  waiting-for  $q$ , 43, 44.  
 $p \mathcal{S} q$ :  $p$  since  $q$ , 43, 45.  
 $p \mathcal{B} q$ :  $p$  back-to  $q$ , 43, 45.  
 $p W q$ :  $p$  while  $q$ , 74.  
 $p \mathcal{P} q$ :  $p$  precedes  $q$ , 74, 308.  
 $\widehat{\Box}, \widehat{\Diamond}, \widehat{\mathcal{U}}, \widehat{\mathcal{W}}$ : strict versions of future temporal operators, 50.  
 $\widehat{\exists}, \widehat{\Diamond}, \widehat{\mathcal{S}}, \widehat{\mathcal{B}}$ : strict versions of past temporal operators, 50.  
 $e^+$ : next value of expression  $e$ , 49.  
 $e^-$ : previous value of expression  $e$ , 49.  
 $p \Rightarrow q$ :  $p$  entails  $q$ , 48.  
 $p[\alpha] \Rightarrow q[\alpha]$ : entailment with instantiation  $\alpha$ , 326.  
 $\Leftrightarrow$ : congruence operator, 48.  
 $s \Vdash p$ : state  $s$  satisfies state formula  $p$ , 42.  
 $(\sigma, j) \models p$ :  $p$  holds at position  $j$  of  $\sigma$ , 43.
- $\Vdash p$ : state formula  $p$  is state valid, 50, 52.  
 $\models p$ : temporal formula  $p$  is valid, 50, 52.  
 $P \Vdash p$ : state formula  $p$  is  $P$ -state valid, 51, 52.  
 $P \models p$ : temporal formula  $p$  is  $P$ -valid, 52.  
 $M \models p$ : temporal formula  $p$  is valid over module  $M$ , 52.  
 $P_i \models_m p$ : temporal formula  $p$  is modularly valid over process  $P_i$ , 52, 336.  
 $p \sim q$ :  $p$  and  $q$  are equivalent, 51.  
 $p \approx q$ :  $p$  and  $q$  are congruent, 51.  
 $p \Rightarrow q \mathcal{W} r$ : simple waiting-for formula, 251.  
 $p \Rightarrow q_m \mathcal{W} (q_{m-1} \cdots (q_1 \mathcal{W} q_0) \cdots)$ : nested waiting-for formula, 48, 251.  
 $p \Rightarrow \Diamond r$ : causality formula, 329.



# General Index

## A

$\alpha$ -closed 443.  
 $\alpha$ -closure 443.  
 $\alpha$ -expansion 444, 446.  
 $\alpha$ -formula 402.  
 $\alpha^{-1}$ -expansion 444, 446.  
 $\alpha$ -table 402, 404, 460.  
accessibility  
  assertion  $acc_P$  221, 223, 224, 225,  
    249.  
  communal 32.  
  requirement 30.  
accessible configuration 33.  
ADD-TWO [program] 92, 114, 116.  
adequate cover 455.  
   $cover_P$  [claim] 455.  
ADEQUATE-SUB [algorithm] 430.  
adequate subgraph 427.  
  check for [algorithm] 430.  
admissible instantiation 326.  
algorithm  
  ADEQUATE-SUB (check for  
    adequate subgraphs) 430.  
  ATOM (atom construction) 406.  
  BEHAVIOR-GRAF (construction  
    of  $B_{(P,\varphi)}$ ) 424.  
  CHECK-NWAIT (checking whether  
    a nested waiting-for formula  
    is  $P$ -valid) 302.  
  CHECK1 (checking for  
     $P$ -invariance) 390.  
  CHECK2 (on-the-fly checking for  
     $P$ -invariance) 390.  
  DECOMPOSE (decompose graph  
    into MSCS's) 299, 315, 415.

garbage collection 244, 250.  
GRAPH (constructing  $\mathcal{A}_{(P,\psi)}$ ) 387.  
INCREMENTAL (incremental  
  tableau construction) 449.  
MIN-RANK (minimal rank  
  determination) 300, 301,  
    303.  
P-SAT (satisfiability of a formula  
  over a program) 433.  
PART-TAB (particle tableau  
  construction) 456, 458, 459,  
    462.  
PRUNE (pruning the tableau)  
  419.  
SAT (satisfiability of a formula)  
  415, 417.  
TABLEAU (tableau construction)  
  407.  
TRANSITION-GRAF (constructing  
  a state-transition graph)  
  228.  
algorithmic  
  construction 201.  
  verification 66, 227, 250, 297,  
    381, 399, 462.  
allocator 180, *see also* resource  
  allocation.  
analysis  
  backward 347.  
  forward 353.  
  reachability 250.  
ancestor 16.  
  common 16.  
  least common (LCA) 16.  
ANY-NAT [program] 67.

- arithmetical completeness 249.  
 arithmetization of boolean values  
     94.  
 array  
     as function 170.  
     find maximum of: *see* program  
         MAX-ARRAY.  
     of semaphores 195.  
     retrieving 171.  
     two-dimensional 224.  
     updating 171.  
 arriving edge 278.  
 assertion 2, 42.  
     characterizing minimal ranks  
         292.  
     conjunctively local 150.  
     control 113.  
     data 113.  
     goal 201.  
     inductive 91.  
     intermediate 164, 165, 254, 260,  
         343.  
     intermittent 77.  
     *P*-state valid 52.  
     state valid 52.  
     local 150.  
     multi-variable 106.  
     primed version of 82.  
     propagation 114, 118, 260.  
     single variable 105.  
     strengthening 99, 111, 257.  
     stronger 91.  
     supplementing 260.  
     transition-validated 105.  
     virtualized 137.  
     weakening 260.  
 assertion language 249.  
     extended 223.  
 assertional verification 249.  
 assignment (multiple) 106, 166.  
 assignment [statement] 5, 20.  
 asynchronous  
     channel 12.  
     communication 55.  
     message passing 75, 184, 232.  
 receive [statement] 21.  
 send [statement] 20.  
 atom 384, 401.  
     conditions for being 461.  
     conditions for [claim] 444.  
     construction [algorithm] 406.  
     fulfills (formula) 410.  
     incomplete: *see* particle.  
     initial 407.  
     necessary conditions for [claim]  
         405.  
     over  $\varphi$  403.  
     (*P*-)reachable 387.  
     predecessor of 407.  
     successor of 407.  
 atom sequence  $\vartheta_\pi$  426.  
 ATOM [algorithm] 406.  
 atomic  
     formula 2.  
     statement 166.  
     step 123.  
 augmentation 144.  
 augmented  
     fair transition system 144.  
     program 144, 147.  
     transition 145.  
 automata  
     counter-free 79.  
      $\omega$ -automata 79.  
     techniques 463.  
 auxiliary variable 144, 147, 150,  
     165, 334, 335, 357.  
     augmenting program 144.  
     essential? 150.  
     sometimes essential 161.  
 auxiliary [mode in program] 147.  
 await [statement] 5, 20, 76.  
 axiomatization (of temporal logic)  
     78.

**B**

- $\beta$ -closed 444.  
 $\beta$ -expansion  
     for atoms 445, 446.  
     for particles 454.

- $\beta$ -expansion (continued)
  - inverse 444, 447, 454.
- $\beta$ -formula 402, 444.
- $\beta$ -table 402, 404, 460.
- $\beta^{-1}$ -expansion 444, 447, 454.
- back-to
  - formula 364, 365.
  - past temporal operator 43, 45, 363.
  - rule 366.
- BACK [rule] 366.
- backward
  - analysis 347, 349.
  - propagation 257, 260, 263.
  - reasoning 342, 348, 349.
- backward propagation [procedure] 261.
- Bakery algorithm 158, 166, 310, *see also* programs MUX-BAK ...
- basic
  - formula 405.
  - invariance [rule] 87, 88, 90, 97, 152.
  - nested waiting-for [rule] 266.
  - node 277.
  - statement 5, 20.
  - type 9.
  - waiting-for [rule] 252, 307.
- before [past temporal operator] 43, 45.
- behavior graph  $\mathcal{B}_{(P,\varphi)}$  424.
  - construction of [algorithm] 424.
  - path in 426.
- BEHAVIOR-GRAF [algorithm] 424.
- BINOM [program] 138, 150.
- BINOM-A [program] 151.
- BINOM-C [program] 139.
- binomial coefficient
  - augmented 151.
  - coarse version 139.
  - fine version 137, 138, 150.
- block [statement] 9.
- body
  - of block 9.
- of linear invariant 202, 207, 213, 217.
- of program 11.
- boolean
  - connective 2, 42.
  - type 2.
  - variable 2.
- bottom-up [approach, method] 102, 104, 165, 201.
- bounded overtaking [property] 182, 166, 182, 251, *see also* overtaking.
  - 1-bounded 264, 265, 270, 274, 279, 282, 370.
  - $1\frac{1}{2}$ -bounded 280.
  - 2-bounded 271, 395.
- branching-time temporal logic 250, 462.
- BUFFER [module] 38.

**C**

- canonical
  - example 248.
  - extended formula 74.
  - formula 58.
  - $\kappa$ -formula 59.
- case splitting (for causality) 348.
- CAUS [rule] 343, 345, 393.
- CAUS-I [rule] 393.
- causality
  - case splitting 348.
  - formula 317, 329.
  - inclusive 393.
  - monotonicity of 348.
  - property 329, 342, 343.
  - rule 342, 343, 344, 345, 393.
  - transitivity of 348.
- ccs language (calculus) 76.
- channel 5.
  - asynchronous 12.
  - declaration of 12.
  - synchronous 12, 21.
  - virtual 334.
- CHECK-NWAIT [algorithm] 302.
- CHECK1 [algorithm] 390.

- CHECK2 [algorithm] 390.  
 children (of compound statement) 7.  
 chopstick exclusion (for dining philosophers) 196, 201.  
 class (of properties) 60, 74.  
 classification 78.  
   diagram 60.  
   of formulas 58.  
   of properties 61.  
   safety liveness 78.  
   safety progress 79.  
 closed  
    $\alpha$ -closed 443.  
    $\beta$ -closed 444.  
   system 36.  
 closure  
    $\alpha$ -closure 443.  
   of formula 401, 462.  
   restricted (for particles) 452.  
   size 402.  
 coarse granularity (of program) 127, 128, 132, 139.  
 COLL-W [rule] 270.  
 collapsing [property] 270.  
 common  
   ancestor 16.  
   factor (of compound node) 278.  
   initial state 68.  
 communal accessibility 32.  
 communication 75.  
   asynchronous 55.  
   predicate 55.  
   statement 5, 10.  
   substatement 234.  
   synchronous 56.  
*comp-expansion* 445, 447.  
 compassion  
   requirement 3, 4.  
   set 27.  
 compassionate  
   subgraph 427.  
   transition 3.  
 compensation expression 131, 134.  
   location-dependent 134.  
     of linear invariant 202, 210, 213, 218.  
     strategy 134.  
 complete (set of formulas) 444.  
 completeness 249.  
   arithmetical 249.  
   by backward view 314.  
   expressive 77.  
   of rule INV-P [theorem] 372.  
   of rule INV [theorem] 221.  
   of rule NWAIT [theorem] 288.  
   relative 78, 249.  
   relative to first-order reasoning 221, 288, 372.  
 completion expansion 445, 447.  
 complexity 462.  
   of Algorithm SAT 417.  
 composite statement 10, 25.  
 composition, parallel 169.  
 compositional verification 39, 79, 336, 337, 396.  
 compound  
   node 277, 278.  
   statement 7, 23, 168.  
 computation 3, 4, 27.  
   divergent 62.  
   modular 40, 336.  
    $P$ -computation 3.  
    $\varphi$ -computation 62.  
   predictive 84, 323.  
    $s$ -computation 68.  
   segment 263, 383.  
   terminating 62.  
 computational  
   induction 165.  
   model 63.  
 CON-I [rule] 89, 90.  
 CONC-W [rule] 270.  
 concatenation [property] 270.  
 concatenation [statement] 7.  
 conclusion (of rule) 54.  
 concurrency 1, 75.  
 concurrent program 127, 164, 249.  
 conditional [statement] 7, 23.  
   one-branch [statement] 7, 23.

- configuration
  - accessible 33.
  - control 33, 128.
- CONFLICT [property] 109.
- conflict-free (location set) 33.
- conflicting labels 16, 109.
- congruence [temporal operator] 48.
- congruent
  - statements 67.
  - temporal formulas 51, 72.
- conjunction [boolean connective]
  - 43.
- conjunctively local 150.
- conjunctiveness (of invariance) 89.
- connected (particles) 453.
- connection requirements 407, 453.
- consecution requirement 4.
- consistent
  - atom with a state 424.
  - locally (set of formulas) 444.
- constant 2.
- construction of
  - behavior graph 424.
  - cover 445.
  - linear invariants 201, 249.
  - reachable-atoms graph 387.
  - state-transition graph 228.
  - tableau 436, 462.
- consume [statement] 7, 23.
- consumer process 131.
- CONT [rule] 349.
- context 68.
- contradiction 349.
- contrary philosopher 198.
- control
  - assertion 113.
  - configuration 33, 128.
  - expression 86.
  - invariant 109.
  - predicate 18.
  - variable 2, 25, 33, 82, 109, 113, 152, 165.
- cooperation [statement] 9, 11, 24, 76.
  - in parameterized program 168.
- variable-size 169.
- correctness 63.
  - partial 62, 92, 106, 138, 153, 164.
  - total 62, 164.
- counter-free automata 79.
- $\text{cover}_P$  [function] 453.
  - adequate cover [claim] 455.
- cover (of formula set) 443.
  - adequate 455.
  - by particles 453.
  - tree representation of 445.
- critical
  - reference 17.
  - section 30.
  - state 94, 283.
  - statement 6, 22.
- CSP language 75.
- CUBE [program] 158.
- cubic power 156, 158.
- cycle
  - in program 207.
  - in verification diagram 275.
  - of cyclic program 213.
- cyclic
  - equations (CE) 213, 242, 243.
  - program 207, 213.
  - distance 284.
- D**
- data
  - assertion 113.
  - precondition 12, 18.
  - relation 25.
  - variable 2, 25, 113.
- data independence 333, 396.
- data-independent program 333, 392.
- data-precondition (of program) 12.
- deadlock
  - freedom [property] 166, 197.
  - prevention 198, 248.
  - state 197.
- declaration
  - local 9.
  - in module 37.

declaration (continued)  
 statement (in block) 9.  
 statement (in program) 11.  
 of channel 12.

**DECOMPOSE** [algorithm] 299, 315,  
 415.

decomposition (of graph into  
 MSCS's) 299, 415.

Dekker's algorithm: *see* programs  
 MUX-DEK . . .

denotational semantics 249.

departing edge 277.

derived temporal operator 74.

detecting termination 237.

development of programs 127.

diagram  
 exit-free 275.  
 INVARIANCE 275, 277.  
 verification 272, 315.  
 WAIT: *see* WAIT diagram.

**DIFF-INC** [program] 163.

diligent transition 3.

**DINE** [program] 196.

**DINE-CONTR** [program] 199.

**DINE-EXCL** [program] 200.

dining philosophers problem 194,  
 195, 198, 248.  
 breaking the symmetry 197.  
 contrary philosopher solution  
 198.  
 one-philosopher excluded  
 solution 198.  
 simple solution 195, 196.

directed  
 edge 424.  
 graph 298.

disabled transition 2.

disjoint successor (of subgraph)  
 299.

disjunction [basic boolean  
 connective] 42, 43.

distributed  
 program 168.  
 termination 250.

divergent computation 62.

**DOUBLE** [program] 204.

drinking philosophers problem 248.

duplicate messages 396.  
 in producer-consumer 333, 356.

**E**

edge  
 arriving 278.  
 departing 277.  
 implicit 272.  
 $\tau$ -edge 272.

elaboration (of parameterized  
 program) 172.

enabled  
 at position 3.  
 continually 3, 4.  
 formula expressing 3.  
 infinitely many times 3, 4.  
 transition 2.

encapsulation convention 277, 315,  
 407, 418.

entailment  
 establishing  $P$ -validity 326.  
 of past formulas 318.  
 reasoning 326.

entailment [temporal operator] 48.

entry  
 label (of process) 9.  
 transition 24.

environment 37, 336.  
 transition 38.

**EQUAL** [property] 109.

equivalence [boolean connective]  
 43.

equivalence [relation]  $\sim_L$  13.

equivalent  
 temporal formulas 51, 72.  
 termination 68.

escape transition 257.

**EVEN** [system] 222, 223.

event  
 receiving 55.  
 sending 55.  
 sending-receiving 57.

- eventually [future temporal operator] 43, 44.  
**EXCH** [program] 163.  
 exclusion  
     mutual: *see* mutual exclusion problem.  
     of temporal miracles 330.  
     requirement 30.  
**existential** [quantifier] 45.  
**exit**  
     label (of process) 9.  
     transition 24.  
**exit-free diagram** 275.  
**expansion**  
      $\alpha$ -expansion 444, 446.  
      $\alpha^{-1}$ -expansion 444, 446.  
      $\beta$ -expansion (for atoms) 445, 446.  
      $\beta$ -expansion (for particles) 454.  
      $\beta^{-1}$ -expansion 444, 447, 454.  
     completion 447.  
     comp-expansion 445, 447.  
     formula 400, 403.  
     inverse 446.  
**expression** 2.  
     compensation 131, 134.  
     defining virtual variable 135.  
     linear 203.  
     next value of 49.  
     previous value of 49.  
**expressive** 249.  
**expressive completeness** 77.  
**expressiveness** 79.  
**extended**  
     canonical formula 74.  
     past formula 74.
- F**
- FACT** [program] 153.  
**FACT** [system] 225.  
**factorial function** 153.  
**failure node** 447.  
**fair**  
     subgraph 427.  
     transition system: *see* FTS.
- fair-merge process 75.  
**FAIR-MERGE** [program] 35.  
 fairness 4, 75.  
 finer granularity 127.  
 finer program: *see* refined program.  
**finite-state**  
     algorithmic verification 227, 297, 381.  
     system 66.  
     verification 250, 399.  
**finite-state program** 227, 228, 250, 297, 381, 422.  
 behavior graph 424.  
 $P$ -invariances of [algorithm] 229.  
 satisfiability over 422.  
 validity over 434.  
**first-order**  
     formula 2.  
     logic 64.  
     property 64.  
     reasoning 64.  
     temporal logic 78.  
     underlying language 2.  
**fixpoint, parallel computation** 241.  
**flexible**  
     quantification 77.  
     system variable 82.  
     variable 43.  
**for** [statement] 8.  
**formula** 2.  
      $\alpha$ -formula 402.  
     atomic 2.  
     back-to 364.  
     basic 405.  
      $\beta$ -formula 402, 444.  
     canonical 58.  
     causality 317, 329.  
     closure of 462.  
     first-order 2.  
     future 46, 455.  
     general 399.  
     guarantee 59.  
      $\kappa$ -formula 59.  
     nested back-to 363, 369.

- formula (continued)  
 nested waiting-for 48, 264, 265,  
 267, 317, 363.  
 obligation 59.  
 past 46, 317.  
 persistence 59.  
 positive form of 452.  
 promising 409.  
 reactivity 59.  
 response 59.  
 safety 59.  
 satisfiability of 461.  
 specifying property 61.  
 state: *see* assertion.  
 state-causality 329.  
 state-quantified 48.  
 temporal: *see* temporal formula.  
 temporally-quantified 48.  
 valid 50, 52, 72.  
 waiting-for 251, 314.
- formulas  
 classification of 58.  
 congruent 51.  
 equivalent 51.  
 mutually satisfiable 404.  
 waiting-for 270.
- forward  
 analysis 353.  
 propagation 257, 258, 263.
- forward propagation [procedure]  
 258.
- framework (for verification) 63.
- FTS (fair transition system) 1, 2, 63,  
 75.  
 augmented 144.  
 computation of 3, 4.  
 run of 4.
- fulfill  
 atom 410.  
 particle 453.
- fulfilling  
 path 410.  
 SCS 413.  
 subgraph 427.  
 tableau 410.
- trail 427.  
 fulfilling path  
 and subgraph [claim] 414.  
 induces models [claim] 411.  
 satisfiability [proposition] 412.  
 fulfillment (of past formulas) 460.  
 function 2.  
 $\alpha$ -closure 443.  
*cover* 444.  
 $cover_P$  453.
- future  
 formula 46, 455.  
 temporal operator 43, 46.
- G**
- GARB-COLL  
 program 245.  
 on the fly 245.
- garbage collection [algorithm] 244,  
 250.
- GCD [program] 14.  
 GCD-F [program] 13.  
 GCDLCM [program] 155.  
 GCDM [program] 154.  
 generalization [heuristic] 112.  
 generalized semaphore 189, 248.  
 global verification 396.
- goal  
 assertion 201.  
 node 272.
- granularity  
 coarse 127.  
 finer 127.
- graph  
 behavior 424.  
 directed 298.  
 reachable-atoms 384.  
 state-transition 228, 250, 297,  
 300, 384.
- GRAPH [algorithm] 387.
- greatest common divisor 12, 13, 14,  
 153, 154, 155.
- grouped [statement] 10, 25.
- guarantee  
 class (of properties) 60.

guarantee (continued)

  formula 59.

guarded command

  language 8.

  statement 76.

## H

halt [statement] 5.

henceforth [future temporal

  operator] 43, 44.

heuristic 257.

  generalization 112.

  range extension 113.

  strengthening 111.

  supplement by precondition 116.

history variable 318, 357, 394.

holds

  at position 45.

  on model 46.

## I

ideal variable 144, 334.

idle statement 6, 22.

idling transition 3, 87, 106, 325.

immediate predecessor 342.

implication [boolean connective]

  43.

implicit edge 272.

implied successor [formula]

  for atom 447.

  for particle 456.

in [mode in declaration] 11.

INC [system] 327, 345, 374, 375,

  376.

INC1 [system] 382, 386, 390.

INC2 [system] 383, 391.

INC-INV [rule] 101.

incomplete atom: *see* particle.

INCREMENT [program] 100.

INCREMENTAL [algorithm] 449.

incremental

  proof 91, 97, 98, 99.

  verification 326.

incremental construction

  of particle tableau 453, 456.

of state-transition graph 228.

  of tableau 443, 449, 463.

individual starvation [property]

  166.

inductive assertion 81, 91.

  finding 104, 167.

  relative to 98.

initial

  atom 385, 407.

  condition 2, 18, 171.

$\varphi$ -atom 408, 424.

$\varphi$ -node 424.

  state 2, 68.

  version (of past formula) 318,  
    319.

initiality

  equations (I) 203, 243.

  requirement 4.

initialization 10.

initialized path 426.

INSENSITIVE [program] 215.

instantiation 326.

  admissible 326.

  tautological 326.

integer division 153, 154.

integer square root: *see* programs

  SQUARE-ROOT · · ·

integer [type] 2.

interference 102, 137.

interleaving 9, 75.

intermediate assertion 77, 164, 165,

    254, 260, 343.

  systematic development of 257.

internal process 11.

interpretation 2.

interval 289.

  empty 251.

$q_i$ -interval 48.

INV [rule] 92, 221.

INV-B [rule] 87, 88, 90, 97, 152.

INV-P [rule] 319, 327, 372.

invalid temporal formula 72.

invariance

  application 167.

  formula 81.

invariance (continued)

modular 340.

property 81, 87.

rules: *see* rules INV . . .

under stuttering 392.

INVARIANCE diagram 275, 277.

claim 275.

invariant 87, 167.

bottom-up method 102, 104, 165.

control 109.

linear (construction) 201.

not inductive 91, 99.

previously established 97.

refinement of 123, 127.

top-down method 104, 111, 165.

inverse

$\alpha$ -expansion 446.

$\beta$ -expansion 447.

verification condition 342, 343.

## J

just

subgraph 427.

transition 3.

justice 75.

requirement 3, 4.

set 27.

## K

$\kappa$ -formula 59.

$\kappa$ -property 61.

KEEPING-UP [program] 40, 338, 392.

## L

label 12.

conflicting 16, 109.

entry and exit 9.

implicitly subscripted 169.

in parameterized program 169.

parallel 16.

language

assertion 42.

CCS 76.

CSP 75.

OCCAM 75.

of guarded commands 8.

simple programming: *see* SPL.

specification 63.

system description 63.

LCA (least common ancestor) 16.

LCR (limited critical reference)

program 17, 123, 137.

restriction 17, 76.

statement 17.

leads from-to 83.

least common ancestor (LCA) 16.

least common multiple 155.

limited critical reference: *see* LCR.

linear

expression 203.

invariant 201, 202.

variable 203.

local

assertion 150.

conjunctively- 150.

declaration 9.

local [mode (in declaration)] 12.

locally consistent 444.

location 13, 105.

conflict-free 33.

corresponding to label 14.

parameterized 174.

range 112.

subscripted 174.

location-dependent expression 134.

locking construct 127.

logic

first-order 64.

modal 76.

temporal: *see* TL.

LOOP [system] 423.

LOOP<sup>+</sup> [system] 432.

## M

*M*-valid 52.

*M*<sub>1</sub> [module] 40, 339.

*M*<sub>2</sub> [module] 41, 339.

matching statements 21.

MAX-ARRAY [program] 173, 177.

- maximal strongly connected
  - subgraph (MSCS) 299, 415.
- maximum of array 173, 177.
- message count 240.
- message duplication 362.
- message passing 5, 75.
  - asynchronous 75, 232.
  - program 34, 78.
  - synchronous 75, 232.
- MIN-RANK [algorithm] 300, 301, 303.
- minimal rank 290, 297, 298.
  - assertion characterizing 292.
  - determination 298, 300, 301, 303.
- MOD [rule] 337.
- modal logic 76.
- mode
  - auxiliary 147.
  - of transition 23, 82.
- mode (in declaration) 11, 12, 37.
- model 43.
  - induce fulfilling paths [claim] 410.
  - induce paths [claim] 409.
  - induced by fulfilling paths [claim] 411.
- model checking 399, 462, 463.
- modular
  - computation 40, 336.
  - invariance 340.
  - validity 52, 337.
- modularity 100.
- module 1, 36, 37, 38, 39, 40, 336.
  - BUFFER 38.
  - $M_1$  40, 339.
  - $M_2$  41, 339.
- modus ponens [rule] 54.
- MON-C [rule] 348.
- MON-I [rule] 89.
- monotonicity
  - of causality 348.
  - of invariance 89.
- MPX-SEM [program] 172, 175.
- MPX-SEM-2 [program] 173.
- MSCS (maximal strongly connected subgraph) 299, 415.
- and satisfiability 416.
- checking fulfillment of 417.
- terminal 419.
- mutual-exclusion problem 29, 69, 70, 166, 248, *see also* programs MUX-... .
- accessibility requirement 30.
- by grouped statements 31.
- by semaphores 30, 93, 98, 104, 168, 229, 434.
- by shared variables 71.
- by synchronous communication 70.
- by test-and-set 161.
- communal accessibility 32.
- exclusion requirement 30.
- for resource allocator 183.
- for  $n$  processes 234.
- generalization of 189.
- with  $M$  semaphores 243.
- with swap 160.
- mutually satisfiable set 405.
- MUX-BAK-A [program] 159, 244, 310.
- MUX-BAK-B [program] 159.
- MUX-BAK-C [program] 160, 310.
- MUX-DEK [program] 126, 155, 309, 310.
- MUX-DEK-A [program] 156, 244, 310.
- MUX-DEK-B [program] 157, 309.
- MUX-PET1 [program] 121, 227, 231, 255, 258, 259, 261, 365, 368, 439, 461.
- MUX-PET2 [program] 124, 268, 271, 274, 276, 278, 305, 308, 350, 370, 395.
- MUX-PET-N [program] 234.
- MUX-SEM [program] 30, 93, 98, 104, 229, 434.
- MUX-SHARED [program] 72.
- MUX-SWAP [program] 161, 310.
- MUX-SYNCH [program] 71.
- MUX-TESTSET [program] 162, 310.
- MUX-VAL-3 [program] 157, 310.
- MUX-WHEN [program] 32.

**N**

*N*-path 298.  
 natural numbers, computing 67.  
**NBACK** [rule] 369, 395.  
 necessity (in modal logic) 76.  
 negation [basic boolean connective]  
     42, 43.  
 nested back-to  
     formula 363, 369.  
     property 395.  
     rule 369.  
 nested unless: *see* nested  
     waiting-for.  
 nested waiting-for  
     algorithm 302.  
     formula 48, 251, 264, 265, 267,  
       289, 317, 363, 395.  
     property 265.  
     rule 264, 267, 285.  
     rule, basic 266.  
 next value (of expression) 49.  
 next [future temporal operator] 43,  
     44, 78.  
 no-invented-messages [property]  
     330, 333, 354, 356, 359.  
 no-invented-replicates [property]  
     333, 356, 362, 392.  
 node  
     basic 277.  
     compound 277.  
     in behavior graph 424.  
     of verification diagram 272.  
      $\varphi$ -node 424.  
     terminal 272.  
 non-interference 165.  
 noncritical  
     section 30.  
     statement 6, 22.  
 nondeterminism 75.  
 nondeterministic  
     program 164.  
     program RES-ND 181.  
 nonescape transition 261.  
**NONSENSE** [program] 208, 212.  
**NWAIT** [rule] 267, 285, 288, 395.

**NWAIT-B** [rule] 266.**O**

$\omega$ -automaton 463.  
 $\omega$ -notation 400.  
 1-bounded overtaking [property]  
     264, 265, 270, 274, 279, 282,  
     370.  
 $1\frac{1}{2}$ -bounded overtaking [property]  
     280.  
 obligation  
     class (of properties) 60.  
     formula 59.  
 OCCAM language 75.  
 offending implication 92.  
 once [past temporal operator] 43,  
     44.  
 one-branch-conditional [statement]  
     7, 23.  
 open system 37.  
 operator: *see* temporal operator.  
 order preservation [property] 251,  
     318, 354, 357, 359, 393.  
 order replication [property] 357,  
     362, 394.  
 out [mode (in declaration)] 12.  
 overtaking analysis 280.  
 overtaking [property] 363.  
     bounded 166, 182, 251.  
     1-bounded 264, 265, 270, 274,  
       279, 282, 370.  
      $1\frac{1}{2}$ -bounded 280.  
     2-bounded 271, 395.  
     unbounded 166.  
 own-in, own-out [mode (in module  
     declaration)] 37.

**P**

$\varphi$ -atom 403, 408.  
 $\varphi$ -node, initial 424.  
 $\varphi$ -reachable SCS 413.  
*P*-accessible state 4, 84.  
*P*-congruence 52.  
*P*-equivalence 52.

- P*-invariance [property] 81, 389.  
 checking [algorithm] 390.  
 of finite-state program  
 [algorithm] 229.  
 on-the-fly checking [algorithm]  
 390.
- P*-invariant 81, 87.  
 previously established 261.
- p*-position (in a model) 45.
- P*-SAT [algorithm] 433.
- P*-satisfiability  
 of formula 432.  
 problem 423.
- P*-segment 288.
- p*-state 42.
- P*-state validity 51, 52, 85, 97.
- P*-validity 51, 52.  
 of entailment 326.  
 of verification diagram 272.  
 problem 423.
- PAR-FIX [program] 242.
- PAR-SUM [program] 169.
- PAR-SUM-E [program] 170.
- parameterization 168.
- parallel  
 composition 169.  
 computation of a fixpoint 242.  
 label 16.  
 statement 16.  
 sum of squares 169, 170.
- PARALLEL [property] 110.
- parameterized program 168, 169,  
 171, 248, 280.  
 cooperation statement 168.  
 elaboration of 172.  
 process 174.  
 selection statement 168.  
 specification of 174.  
 statement 169.  
 system variable in 170.  
 verification of 175.
- parameterized transition systems:  
*see* parameterized program.
- parbegin [statement] 76.
- parity [property] 79.
- PART-TAB [algorithm] 456, 458, 459,  
 462.
- partial correctness [property] 62,  
 92, 106, 138, 153, 164.
- partial substitution 86.
- particle (incomplete atom) 451,  
 452.  
 connected to 453.  
 fulfill (formula) 453.  
 realizable 455.
- particle cover 453.
- particle tableau  $\tilde{T}_\varphi$  451, 456, 458.  
 construction [algorithm] 456.  
 construction [claim] 458.
- past  
 extended formula 74.  
*formula: see past formula.*  
 statifying 357.  
 temporal operator 43, 46.
- past formula 46, 317.  
 entailment of 318.  
 fulfillment of 460.  
 initial version of 318, 319.  
 previous-free 325.  
 primed version of 318, 321, 324.  
 statification of 373.  
 verification condition 318.
- past invariance rule 327, 344.
- past verification condition 323, 324.
- past waiting-for [rule] 366.
- path  
 fulfilling 410.  
 in behavior graph 426.  
 in tableau 408.  
 induced by model 409.  
 initialized 426.  
 $N$ -path 298.
- Peano arithmetic 249.
- periodic sequence 400.
- persistence  
 class (of properties) 60.  
 formula 59.
- Peterson's algorithm  
 coarse: *see* program MUX-PET1.  
 for  $n$  processes 234.

- Peterson's algorithm (continued)  
 refined: *see* program MUX-PET2.
- philosophers  
 dining [problem] 194.  
 thinking 248.
- PMUX-MAN [program] 233.
- positive form (of formula) 452.
- possibility (in modal logic) 76.
- post-label (of statement) 15.
- post-location (of statement) 15.
- postcondition 61, 258.
- PPS (principally past subformulas) 373.
- PREC [rule] 308.
- precedence [property] 251, 255, 314, 368, 393.
- precedence [temporal operator] 74, 308, 314.
- precondition 61, 114, 152, 260.  
 data 12, 18.  
 supplement by 116.  
 weakest 166.
- predecessor  
 of atom 407.  
 $\tau$ -predecessor 114, 260.
- predicate  
 communication 55.  
 control 18.
- predictive computation 84, 323.
- prefer-to [statement] 183.
- prefix inclusion [property] 361.
- premises (of rule) 54.
- previous-free past formula 325.
- previous value (of expression) 49.
- previous value [temporal operator] 72.
- previously established  
 invariant 97.  
 $P$ -invariant 261.
- previously [past temporal operator] 43, 45.
- PRIME [program] 57.
- prime numbers, computation of 57.
- primed version  
 of assertion 82.
- of past formula 318, 321, 324.  
 of variable 3.
- principally past subformulas (PPS) 373.
- priority 120, 125.
- problem  
 dining philosophers: *see* dining philosophers problem.  
 mutual exclusion: *see* mutual-exclusion problem.  
 producer-consumer: *see* producer-consumer problem.  
 readers writers: *see* readers-writers problem.
- process  
 consumer 131.  
 fair-merge 75.  
 in cooperation statement 9.  
 internal 11.  
 needing resource: *see* program RES-3.
- parameterized 174.
- priority 120.
- producer 131.  
 represented by module 39.  
 top-level 11, 41, 52, 109.
- PROD-CONS [program] 34, 133, 313, 330, 331, 346, 354, 392, 393, 395.
- PROD-CONS-A [program] 336.
- PROD-CONS-C [program] 131.
- PROD-CONS-H [program] 358.
- PROD-CONS-S [program] 162.
- PROD-CONS-SV [program] 216, 243.
- produce [statement] 6, 22.
- producer-consumer problem 34, 166, 331, *see also* programs PROD-CONS...  
 coarse version 131.  
 duplicate messages 333, 356, 362.  
 refined version 133.  
 with asynchronous channels 313.  
 with auxiliary variable 336.

- producer-consumer problem  
(continued)  
with history variables 358.  
with shared variables 162, 216,  
243.
- program 11.  
ADD-TWO 88, 92, 114, 116.  
ANY-NAT 67.  
augmented (by auxiliary  
variables) 144.  
BINOM 138, 150.  
BINOM-A 151.  
BINOM-C 139.  
body of 11.  
coarse 127, 128, 132, 139.  
concurrent 127, 164, 249.  
CUBE 158.  
cyclic 207, 213.  
data-independent 333, 392.  
development of 127.  
DIFF-INC 163.  
DINE 196.  
DINE-CONTR 199.  
DINE-EXCL 200.  
distributed 168.  
DOUBLE 151, 202, 204.  
EXCH 163.  
FACT 153.  
FAIR-MERGE 35.  
finer: *see* refined program.  
finite-state 228, 250, 297, 381,  
422.  
GARB-COLL 245.  
GCD 14.  
GCD-F 13.  
GCDLCM 155.  
GCDM 154.  
IDIV 154.  
INCREMENT 100.  
INSENSITIVE 215.  
invariant of 87, 167.  
KEEPING-UP 40, 338, 392.  
LCR 17, 123, 137.  
MAX-ARRAY 173, 177.  
message passing 34, 78.
- MPX-SEM 172, 175.  
MPX-SEM-2 173.  
MUX-BAK-A 159, 244, 310.  
MUX-BAK-B 159.  
MUX-BAK-C 160, 310.  
MUX-DEK 126, 155, 309, 310.  
MUX-DEK-A 156, 244, 310.  
MUX-DEK-B 157, 309.  
MUX-PET<sub>1</sub> 121, 227, 231, 255, 258,  
259, 261, 365, 368, 439, 461.  
MUX-PET<sub>2</sub> 124, 268, 271, 274, 276,  
278, 305, 308, 350, 370, 395.  
MUX-PET-N 234.  
MUX-SEM 30, 93, 98, 104, 229,  
230, 434.  
MUX-SHARED 72.  
MUX-SWAP 161, 310.  
MUX-SYNCH 71.  
MUX-TESTSET 162, 310.  
MUX-VAL-3 157, 310.  
MUX-WHEN 32.  
nondeterministic 164.  
NONSENSE 208, 212.  
PAR-FIX 242.  
PAR-SUM 169.  
PAR-SUM-E 170.  
parameterized 168, 169, 171, 248,  
280.  
PMUX-MAN 233.  
PRIME 57.  
PROD-CONS 34, 133, 313, 330,  
331, 346, 354, 392, 393, 395.  
PROD-CONS-A 336.  
PROD-CONS-C 131.  
PROD-CONS-H 358.  
PROD-CONS-S 162.  
PROD-CONS-SV 216, 243.  
property of 54, 61.  
READ-WRITE 190, 248.  
refined 118, 123, 128, 132, 166.  
RES-3 192.  
RES-MP 182, 232, 248, 282.  
RES-ND 181, 232.  
RES-SEM 179.  
RES-SV 185, 248, 311.

- program (continued)  
   RES-SV-S 312.  
   RING-TERM 238, 240.  
   run of 4.  
   SAMPLE1 28.  
   SAMPLE2 28.  
   SAMPLE3 29.  
   SB 67.  
   SEM-3 210, 212, 213.  
   SEM-N 243.  
   semantics of 18.  
   sequential 127.  
   SERV-RING 236.  
   SQUARE-ROOT 106, 118.  
   SQUARE-ROOT-A 149.  
   SQUARE-ROOT-R 129, 136, 148.  
   SQUARE-ROOT\* 128, 134.  
   SUM 112.  
   syntax 11.  
   terminating 61.  
   TRY-MUX1 69.  
   TURN 70.  
   variable 12, 18.  
   verification: *see* verification.  
 programming language, simple: *see*  
   SPL.  
 programs, family of 168.  
 progress (class of properties) 60.  
 promising formula 410.  
 proof 64.  
   backward analysis 349.  
   by contradiction 349.  
   lattice 315.  
   obligation: *see* verification  
     condition.  
   rule 54, 64.  
 proof system 64.  
   Hoare-style 165.  
 propagation  
   backward 257, 260.  
   forward 257, 258.  
   forward vs. backward 262.  
   of assertion 114, 118.  
   technique 128.  
 properties  
   class of 74.  
   classification of 61.  
 property 54, 61.  
   bounded overtaking 166, 182,  
     251.  
   1-bounded overtaking 264, 265,  
     270, 274, 279, 282, 370.  
    $1\frac{1}{2}$ -bounded overtaking 280.  
   2-bounded overtaking 271, 395.  
   causality 329, 342, 343.  
   collapsing 270.  
   CONFLICT 109.  
   deadlock freedom 166, 197.  
   EQUAL 109.  
   first-order 64.  
   individual starvation 166.  
   invariance 87.  
    $\kappa$ -property 61.  
   nested back-to 395.  
   nested waiting-for 265.  
   no-invented-messages 330, 333,  
     354, 356, 359.  
   no-invented-replicates 333, 356,  
     362, 392.  
   order preservation 251, 318, 354,  
     357, 359, 393.  
   order replication 357, 362, 394.  
   overtaking: *see* overtaking.  
   PARALLEL 110.  
   parity 79.  
   precedence 251, 255, 314, 368,  
     393.  
   prefix inclusion 361.  
   safety 64.  
   simple precedence 255.  
   SOMEWHERE 109.  
   specification of 54, 61.  
   starvation freedom 197, 200, 248.  
   state causality 329.  
   termination: *see* termination.  
   propositional  
     dynamic logic (PDL) 462.  
     temporal logic 77.  
   PRUNE [algorithm] 419.  
   pruned tableau 419.

**Q**

*q*-interval 252.  
 quantification 2, 77.  
   flexible 77.  
   rigid 77.  
 quantifier 42.  
   existential 45.  
   restriction about 48.  
   universal 45.

**R**

*r*-free (computation segment) 263.  
 range extension [heuristic] 113.  
 rank  
   minimal 290, 298.  
   of state 290.  
 reachability analysis 250.  
 reachable

$\varphi$ -reachable scs 413.  
   atom 387.  
 reachable-atoms graph 384, 387,  
   390, 391.  
   construction: *see* algorithm  
     GRAPH.

reactive  
   program 1.  
   system 64.

reactivity  
   class (of properties) 60.  
   formula 59.

READ-WRITE [program] 190, 248.  
 readers-writers problem 189, 190,  
   191, 248, *see also* program  
     READ-WRITE.

reading reference 17.

realizable particle 455.

reasoning  
   backward 342, 348, 349.  
   first-order 64.

receive [statement] 6.

receiving event 55.

recurrence equation 107.

reference

  critical 17.

  reading 17.

writing 17.

refined program 123, 128, 132, 166.

refinement 166.

  of invariant 123, 127.

  of programs 81.

REFL-WAIT [rule] 307.

region (set of locations) 102.

relative

  completeness 78, 249, 372.

  inductive assertion 98.

release [statement] 6, 22.

replacement 326.

replication: *see* no-invented-replicates, order replication.

request [statement] 6, 21, 159.

RES-3 [program] 192.

RES-MP [program] 182, 232, 248,  
   282.

RES-ND [program] 181, 232.

RES-SEM [program] 179.

RES-SV [program] 185, 248, 311.

RES-SV-S [program] 312.

resource allocation 179.

  bounded overtaking 182.

  by message passing 180, 182, 282.

  by semaphores 179.

  by shared variables 184, 185, 311.

  by token ring 235.

  for three processes 191, 192.

  multiple 191.

  nondeterministic message

    passing 180, 181.

  overtaking analysis of 280.

  single 179, 248.

  with single array 312.

response

  class (of properties) 60.

  formula 59.

rewriting rule 452.

right constant (of linear invariant)

  202, 212, 213, 218.

rigid

  quantification 77.

  specification variable 82.

  variable 43.

RING-TERM [program] 238, 240.  
 rule  
   BACK 366.  
   CAUS 343, 345, 393.  
   CAUS-I 393.  
   causality 342.  
   COLL-W 270.  
   CON-I 89, 90.  
   CONC-W 270.  
   conclusion of 54.  
   CONT 349.  
   INC-INV 101.  
   INV 92, 221.  
   INV-B 87, 88, 90, 97, 152.  
   INV-P 319, 327, 372.  
   MOD 337.  
   modus ponens 54.  
   MON-C 348.  
   MON-I 89.  
   NBACK 369, 395.  
   NWAIT 267, 285, 288, 395.  
   NWAIT-B 266.  
   PREC 308.  
   premises of 54.  
   proof 64.  
   REFL-WAIT 307.  
   reflexive version 307.  
   SPL-C 348.  
   SV-PSV 97.  
   TRN-C 348.  
   verification 66.  
   WAIT 254, 307.  
   WAIT-B 252, 307.  
   WAIT-P 366.  
 run 4.

**S**

safety  
   class (of properties) 60.  
   formula 59.  
   general [property] 317.  
   property 64.  
 safety-liveness classification 78.  
 safety-progress classification 79.  
 safety property 81, 92.

SAMPLE1 [program] 28.  
 SAMPLE2 [program] 28.  
 SAMPLE3 [program] 29.  
 SAT [algorithm] 415, 417.  
 satisfiability  
   and adequate SCS 428.  
   and fulfilling paths [proposition] 412.  
   and MSCS [proposition] 416.  
   of formula [algorithm] 415.  
   of temporal formula 50, 399, 458, 461.  
   over finite-state program 422.  
   over program [algorithm] 433.  
   over system 461.  
 satisfiable (temporal formula) 50.  
 SB [program] 67.  
 schematic statement 6, 22.  
 SCS (strongly connected subgraph) 315, 413.  
   fulfilling 413.  
   maximal (MSCS) 299, 415.  
    $\varphi$ -reachable 413.  
   satisfiability and adequate [proposition] 298, 428.  
   transient 413.  
 segment 288.  
   originates at 288.  
   satisfaction [claim] 289.  
 selection [statement] 8, 76.  
   in parameterized program 168.  
 self-disabling transition 26.  
 SEM-3 [program] 210, 212, 213.  
 SEM-N [program] 243.  
 semantics  
   denotational 249.  
   of module 38.  
   of SPL programs 18.  
   of temporal formulas 43.  
   possible worlds 76.  
   tableaux 462.  
 semaphore 75, 76, 127, 137, 166, 248.  
   array of 195.  
   generalized 189.

- semaphore (continued)  
 mutual exclusion by 93, 98.  
 resource allocation by 179.  
 statement 21.
- send [statement] 5.  
 send-receive [statement] 21.  
 sending event 55.  
 sending-receiving event 57.  
 sequential program 127, 164.  
 SERV-RING [program] 236.
- set (of formulas)  
 $\alpha$ -closed 443.  
 $\beta$ -closed 444.  
 complete 444.  
 locally consistent 444.
- set (of particles) 453.  
 set partitioning 163.  
 shared variables 75, 156.  
 producer-consumer with 216.  
 resource allocation by 184, 311.
- simple precedence [property] 255.  
 simple programming language: *see* SPL.
- simple waiting-for [formula] 251.
- simplification  
 by partial substitution 86.  
 of control expressions 86.  
 of temporal formulas 451.
- since [past temporal operator] 43,  
 45, 77.
- size of  
 formula 402.  
 tableau 460.
- skip [statement] 5, 20.
- so-far [past temporal operator] 43,  
 44.
- SOMEWHERE [property] 109.
- sorted decomposition (of  
 subgraphs) 299, 300, 305.
- specification  
 language 63.  
 of parameterized program 174.  
 of property 54.  
 temporal 64.  
 variable 82.
- SPL (simple programming language) 5, 63.  
 semantics 18.  
 syntax 5.
- SPL-C [rule] 348.
- splitting states 381.
- SQUARE [system] 145.
- SQUARE-ROOT [program] 106, 118.
- SQUARE-ROOT-A [program] 149.
- SQUARE-ROOT-R [program] 129,  
 136, 148.
- SQUARE-ROOT\* [program] 128, 134.
- star-free regular expression 79.
- starvation freedom [property] 197,  
 200, 248.
- starvation, individual [property] 166.
- state 2.  
 critical 94, 283.  
 deadlock 197.  
 initial 2.  
 minimal rank of 297.  
 $p$ -state 42.  
 pre-critical 283.  
 rank of 290.  
 satisfies assertion 42.  
 splitting 381.  
 $\tau$ -successor of 2.  
 terminal 62, 68.  
 $U$ -variant of 4.  
 variable 2.
- state-causality  
 formula 329.  
 property 329.
- state formula: *see* assertion.
- state-quantified [formula] 48.
- state sequence  $\sigma_\pi$  426.
- state space, partition of 94.
- state-transition graph 228, 250,  
 297, 300, 304, 306, 384, 424.  
 for program MUX-PET1 231.  
 for program MUX-SEM 230.  
 incremental construction 228.
- state validity 50, 97.
- statecharts [visual language] 315.

statement 5, *see also* Programming Language (in Index to Symbols).  
 ancestor of 16.  
 assignment 5, 20.  
 asynchronous receive 21.  
 asynchronous send 20.  
 atomic 166.  
 await 5, 20, 76.  
 basic 5, 20.  
 block 9.  
 common ancestor of 16.  
 communication 5.  
 composite 10, 25.  
 compound 7, 23, 168.  
 concatenation 7.  
 conditional 7, 23.  
 consume 7, 23.  
 cooperation 9, 11, 24, 76.  
 critical 6, 22.  
 declaration (in block) 9.  
 declaration (in program) 11.  
 for 8.  
 grouped 10, 25.  
 halt 5.  
 idle 6, 22.  
 in parameterized program 169.  
 LCR 17.  
 noncritical 6, 22.  
 one-branch-conditional 7, 23.  
 parallel 16.  
 parbegin 76.  
 post-label of 15.  
 post-location of 15.  
 prefer-to 183.  
 produce 6, 22.  
 receive 6.  
 release 6, 22.  
 request 6, 21, 159.  
 schematic 6, 22.  
 selection 8, 76.  
 semaphore 21.  
 send 5.  
 send-receive 21.  
 skip 5, 20.

stronger 91.  
 swap 159.  
 synchronous send-receive 21.  
 test-and-set 161.  
 when 8, 76.  
 while 8, 24.  
 statements  
   congruent 67.  
   matching send-receive 21.  
 static initialization 10.  
 stratification 396.  
   of past formula 357, 373.  
 strategy  
   compensation expression 134.  
   incremental proof 97.  
   stronger assertion 91.  
 strengthening  
   heuristic 111.  
   of assertion 91, 111, 257.  
 strict version (of temporal operators) 50, 77.  
 stronger  
   assertion 91.  
   statement 91.  
 strongly connected subgraph: *see* SCS.  
   maximal: *see* MSCS.  
 strongly-fair transition 3.  
 structured type 9.  
 stuttering 392.  
   sensitive to 78.  
   transition 3.  
 subgraph  
   adequate 427.  
   and fulfilling path [claim] 414.  
   compassionate 427.  
   fair 427.  
   fulfilling 427.  
   just 427.  
   maximal strongly connected: *see* MSCS.  
   strongly connected: *see* SCS.  
   transition taken in 427.  
 subscript 169.  
 SUB2 [system] 303.

- succession of intervals 251.  
 successor  
   disjoint 299.  
   of atom 385, 407, 449, 461.  
   of particle 455.  
    $\tau$ -successor (of state) 2.  
 sum of squares 169.  
 SUM [program] 112.  
 supplementing  
   an assertion 260.  
   by precondition [heuristic] 116.  
 supported  $\beta$ -formula 444.  
 SV-PSV [rule] 97.  
 swap [statement] 159.  
 synchronization 75.  
   construct 127, 248.  
 synchronous  
   channel 12, 21.  
   communication 56.  
   message passing 75, 180, 184,  
     232.  
   send-receive [statement] 21.  
 syntax  
   module 37.  
   parameterized program 168.  
   program 11.  
   SPL 5.  
 SYS-A [system] 290.  
 SYS-B [system] 290, 292, 294.  
 system  
   closed 36.  
   description language 63.  
   EVEN 222, 223.  
   FACT 225.  
   fair transition: *see* FTS.  
   finite-state 66.  
   INC 327, 345, 374, 375, 376.  
   INC1 382, 386, 390.  
   INC2 383, 391.  
   LOOP 423.  
   LOOP<sup>+</sup> 432.  
   open 37.  
   reactive 64.  
   satisfiability over 461.  
   SQUARE 145.
- SQUARE 145.  
 SUB2 303.  
 SYS-A 290.  
 SYS-B 290, 292, 294.  
 variable 2, 18, 170.
- T**
- $\tau$ -edge 272.  
 $\tau$ -modifiable variable 86.  
 $\tau$ -predecessor 114, 260.  
 $\tau$ -preserved variable 86.  
 $\tau$ -successor 427.  
 2-bounded overtaking 271, 395.  
 table  
    $\alpha$ -table 402, 404, 460.  
    $\beta$ -table 402, 404, 460.  
 tableau 407, 424, 451, *see also*  
   particle tableau.  
   construction of 407, 451, 462,  
     463.  
   fulfilling 410.  
   incremental construction of 443,  
     449, 463.  
   pruned 419.  
   size of 460.  
 tableau (particles) 451.  
   construction of 456.  
 TABLEAU [algorithm] 407.  
 tautological instantiation 326.  
 temporal formula 42.  
   holds at position 45.  
   holds on model 46.  
   invalid 72.  
    $P$ -valid 52.  
   satisfiability 400.  
   satisfiable 50.  
   semantics of 43.  
   simplification of 451.  
   valid 50, 52, 72.  
   validity 400.  
 temporal formulas  
   congruent 51, 72.  
   equivalent 51, 72.  
 temporal instantiation 326.  
 temporal logic: *see* TL.

- temporal operator 42.
  - congruence 48.
  - derived 74.
  - entailment 48.
  - future 43, 46.
  - next 78.
  - past 43, 46.
  - precedence 74, 308, 314.
  - previous value 72.
  - strict version 50, 77.
  - while 74.
- temporal replacement 326.
- temporal specification 64.
- temporal tableau 399.
- terminal
  - location set 62.
  - MSCS 419.
  - node 272.
  - state 62, 68.
- terminating
  - computation 62.
  - $\varphi$ -computation 62.
  - program 61.
- termination [property] 61, 63, 92.
  - asynchronous 240.
  - detecting 237.
  - distributed 237, 250.
  - equivalent 68.
  - global 239.
  - of sequential program 164.
- test-and-set [statement] 161.
- tie breaking 120, 125.
- time modality 76.
- TL (temporal logic) 1, 42, 63, 76.
  - axiomatization 78.
  - branching-time 250.
  - first-order 78.
  - future fragment of 77.
  - propositional 77.
- top-down [approach, method] 102, 104, 111, 165.
- top-level process 11, 41, 52, 109.
- total correctness 62.
  - of sequential program 164.
- trail (path in tableau) 423.
- fulfilling 427.
- satisfiability and fulfillment [proposition] 427.
- transient scs 413.
- transition 2, 19.
  - augmented 145.
  - compassionate 3.
  - diligent 3.
  - disabled on state 2.
  - enabled at position 3.
  - enabled on state 2.
  - enabledness formula 3.
  - entry and exit 24.
  - environment 38.
  - escape 257.
  - idling 3, 87, 106, 325.
  - just 3.
  - nonescape 261.
  - of parameterized program 170, 171.
  - potentially falsifying 102.
  - potentially validating 102.
  - preserves assertion 102.
  - self-disabling 26.
  - strongly fair 3.
  - stuttering 3.
  - taken at position 3.
  - weakly fair 3.
- transition equations (T) 203, 243.
- TRANSITION-GRAF [algorithm] 228.
- transition relation 3.
  - of conditional 82.
  - of parameterized program 170, 171.
  - of while 82.
- transition semantics (of program) 18.
- transition system 1, 2.
  - fair: *see* FTS.
  - parameterized: *see* parameterized program.
- transition-validated assertion 105.
- transitivity of causality 348.

tree computation (of cover) 445,  
 450.  
 by particles 454.  
 TRN-C [rule] 348.  
 TRY-MUX1 [program] 69.  
 TURN [program] 70.  
 2-bounded overtaking 271, 395.  
 type (of variable) 2, 9.  
 type-consistent interpretation 2.

**U**

*U*-variant (of state) 4.  
 unbounded overtaking [property] 166.  
 universal quantifier 45.  
 unless: *see* waiting-for.  
 unprimed variable 3.  
 until [future temporal operator] 43,  
 44, 77.  
 updating (an array) 171.

**V**

valid  
*M*-valid 52.  
 modularly 52, 337.  
*P*-state valid 51, 52, 85, 97.  
*P*-valid 51, 52.  
 state valid 50.  
 temporal formula 50, 52, 72.  
 verification diagram 272.  
 validity  
   of temporal formula 399.  
   over finite-state program 434.  
   over finite-state system 399.  
 validity transfer [claim] 146.  
 variable  
   auxiliary 144, 147, 150, 165, 334,  
     335, 357.  
   boolean 2.  
   control 2, 25, 33, 82, 109, 113,  
     152, 165.  
   data 2, 25, 113.  
   flexible 43.  
   history 318, 357, 394.  
   ideal 144, 334.

linear 203.  
 primed 2, 3.  
 program 12, 18.  
 rigid 43.  
 shared 75, 156.  
 state 2.  
 system 2, 18, 170.  
 $\tau$ -modifiable 86.  
 $\tau$ -preserved 86.  
 type of 2.  
 unprimed 3.  
 virtual 135, 334, 360.  
 variant (of state) 4.  
 verification  
   algorithmic: *see* algorithmic verification.  
   assertional 249.  
   compositional 39, 79, 336, 337,  
     396.  
   finite-state 250, 399.  
   framework for 63.  
   global 396.  
   incremental 326.  
   methodology 318.  
   of parameterized programs 175.  
   of programs 164.  
   rule 66.  
   technique 63.  
 verification condition 82, 84, 152.  
   associated with verification diagram 272.  
   for idling transition 87.  
   for mode of transition 82.  
   for past formulas 318, 323, 324.  
   inverse 342, 343.  
   simplified version 86.  
   trivial 85, 101.  
 verification diagram 272, 315.  
   associated verification conditions 272.  
   INVARIANCE diagram 275.  
*P*-valid 272.  
 valid 272.  
 WAIT diagram 273.

- virtual  
channel 334.  
variable 135, 334, 360.
- virtualized version of assertion 137.
- vocabulary 2.
- W**
- WAIT [rule] 254, 307.  
WAIT-B [rule] 252, 307.  
WAIT-P [rule] 366.  
WAIT diagram 273, 287, 310.  
for 1-bounded overtaking 279.  
waiting-for [formula] 251, 270, 314.  
nested 48.  
simple 251.  
waiting-for [future temporal  
operator] 43, 44, 314, 363.
- waiting-for [rule]  
basic 251.  
general 253.
- weakening (of assertion) 260.
- weakest precondition 166.
- weakly-fair transition 3.
- when [statement] 8, 76.
- where clause (in declaration) 9.
- while  
irreproducible 66.  
statement 24.  
temporal operator 74.
- while [statement] 8.
- writing reference 17.

Manna  
Pnueli  
Temporal  
Logic

A reactive system is a system that maintains an ongoing interaction with its environment. The family of such systems includes concurrent programs, as well as a variety of other programs whose correct and reliable construction is particularly important: air traffic control systems, controllers for mechanical devices such as trains, and ongoing processes such as nuclear reactors. The approach taken in this book is to present verification methods for proving that such reactive systems meet their specifications, expressed as safety properties in the language of temporal logic. The text includes deductive methods based on theorem proving, as well as automatic methods based on model checking.

Researchers and students interested in the analysis and verification of reactive systems will find this book to be a comprehensive guide to how