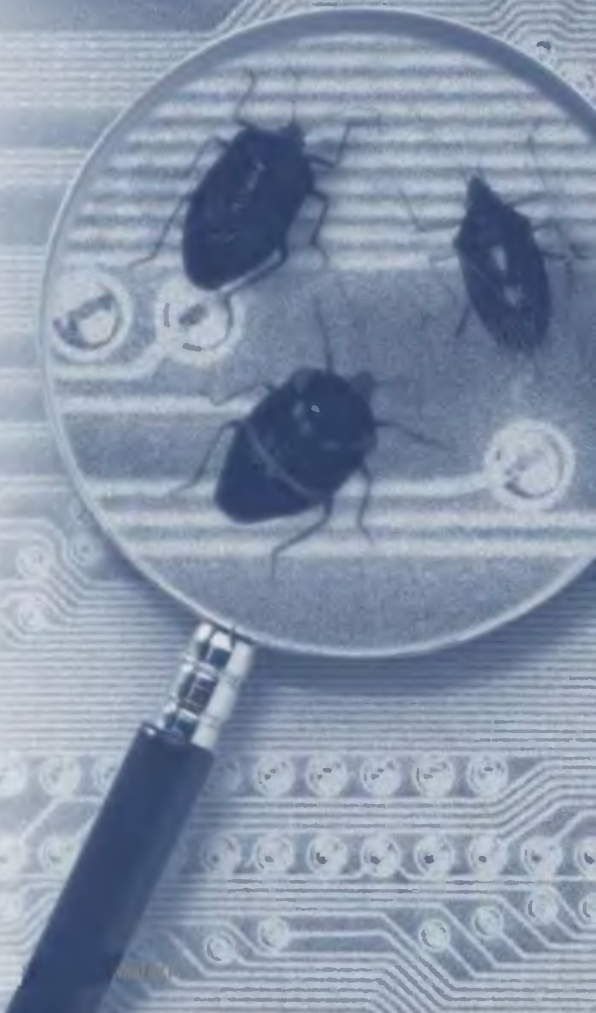
Licensed to Ivo Willemsen

Model Checking



Edmund M. Clarke, Jr.,
Orna Grumberg,
and Doron A. Peled

Model Checking

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled

The MIT Press
Cambridge, Massachusetts
London, England

Second printing, 2000

© 1999 Edmund M. Clarke, Jr., Orna Grumberg, and Lucent Technologies

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set in Times Roman by Windfall Software using ZzTEX and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Clarke, E. M., 1945–
Model checking / Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-03270-8 (hc: alk. paper)
1. Computer systems—Verification. I. Grumberg, Orna.
II. Peled, Doron, 1962–. III. Title.
QA76.76.V47C553 1999
004.2′1—dc21 99-17979
CIP

To
Martha, Selden, Jonathan, and Jefferey
Manfred, Noa, and Hila
Hana and Uri Feld and Dvora and Yaakov Zumer

Contents

Foreword by Amir Pnueli xi
Preface xiii

1 Introduction 1
1.1 The Need for Formal Methods 1
1.2 Hardware and Software Verification 2
1.3 The Process of Model Checking 4
1.4 Temporal Logic and Model Checking 4
1.5 Symbolic Algorithms 6
1.6 Partial Order Reduction 8
1.7 Other Approaches to the State Explosion Problem 10

2 Modeling Systems 13
2.1 Modeling Concurrent Systems 14
2.2 Concurrent Systems 17
2.3 Example of Program Translation 24

3 Temporal Logics 27
3.1 The Computation Tree Logic CTL* 27
3.2 CTL and LTL 30
3.3 Fairness 32

4 Model Checking 35
4.1 CTL Model Checking 35
4.2 LTL Model Checking by Tableau 41
4.3 CTL* Model Checking 46

5 Binary Decision Diagram 51
5.1 Representing Boolean Formulas 51
5.2 Representing Kripke Structures 57

6 Symbolic Model Checking 61
6.1 Fixpoint Representations 61
6.2 Symbolic Model Checking for CTL 66
6.3 Fairness in Symbolic Model Checking 68
6.4 Counterexamples and Witnesses 71
6.5 An ALU Example 75
6.6 Relational Product Computations 77
6.7 Symbolic LTL Model Checking 87

7 Model Checking for the μ-Calculus 97
7.1 Introduction 97
7.2 The Propositional μ-Calculus 98
7.3 Evaluating Fixpoint Formulas 101
7.4 Representing μ-Calculus Formulas Using OBDDs 104
7.5 Translating CTL into the μ-Calculus 107
7.6 Complexity Considerations 108

8 Model Checking in Practice 109
8.1 The SMV Model Checker 109
8.2 A Realistic Example 112

9 Model Checking and Automata Theory 121
9.1 Automata on Finite and Infinite Words 121
9.2 Model Checking Using Automata 123
9.3 Checking Emptiness 129
9.4 Translating LTL into Automata 132
9.5 On-the-Fly Model Checking 138
9.6 Checking Language Containment Symbolically 139

10 Partial Order Reduction 141
10.1 Concurrency in Asynchronous Systems 142
10.2 Independence and Invisibility 144
10.3 Partial Order Reduction for LTL_{-X} 147
10.4 An Example 151
10.5 Calculating Ample Sets 154
10.6 Correctness of the Algorithm 160
10.7 Partial Order Reduction in SPIN 164

11 Equivalences and Preorders between Structures 171
11.1 Equivalence and Preorder Algorithms 178
11.2 Tableau Construction 180

12 Compositional Reasoning 185
12.1 Composition of Structures 187
12.2 Justifying Assume-Guarantee Proofs 189
12.3 Verifying a CPU Controller 190

13 Abstraction 193
13.1 Cone of Influence Reduction 193
13.2 Data Abstraction 195

14 Symmetry 215
14.1 Groups and Symmetry 215
14.2 Quotient Models 218
14.3 Model Checking with Symmetry 221
14.4 Complexity Issues 224
14.5 Empirical Results 228

15 Infinite Families of Finite-State Systems 231
15.1 Temporal Logic for Infinite Families 231
15.2 Invariants 232
15.3 Futurebus+ Example Reconsidered 235
15.4 Graph and Network Grammars 238
15.5 Undecidability Result for a Family of Token Rings 248

16 Discrete Real-Time and Quantitative Temporal Analysis 253
16.1 Real-Time Systems and Rate-Monotonic Scheduling 253
16.2 Model Checking Real-Time Systems 254
16.3 RTCTL Model Checking 255
16.4 Quantitative Temporal Analysis: Minimum/Maximum Delay 256
16.5 Example: An Aircraft Controller 259

17 Continuous Real Time 265
17.1 Timed Automata 265
17.2 Parallel Composition 268
17.3 Modeling with Timed Automata 269
17.4 Clock Regions 274
17.5 Clock Zones 280
17.6 Difference Bound Matrices 287
17.7 Complexity Considerations 291

18 Conclusion 293

References 297
Index 309

Foreword

It is widely agreed that the main obstacle to "help computers help us more" and relegate to these helpful partners even more complex and sensitive tasks is not inadequate speed and unsatisfactory raw computing power in the existing machines, but our limited ability to design and implement complex systems with sufficiently high degree of confidence in their correctness under all circumstances.

This problem of *design validation*—ensuring the correctness of the design at the earliest stage possible—is a major challenge in any responsible system development process, and the activities intended for its solution occupy an ever increasing portions of the development cycle cost and time budgets.

The currently practiced methods for design validation in most sites are still the veteran techniques of *simulation* and *testing*. Although provably effective in the very early stages of debugging, when the design is still infested with multiple bugs, their effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs. A serious problem with these techniques is that one is never sure when they have reached their limits or even an estimate of how many bugs may still lurk in the design. As the complexity of designs drastically increases, say from having .5 million gates per chip to advanced designs with 5 million gates per chip, some far-seeing managers foresee the complete collapse of these conventional methods and their total inability to scale up.

A very attractive and increasingly appealing alternative to simulation and testing is the approach of *formal verification,* which is the main topic of this book. While simulation and testing explore *some* of the possible behaviors and scenarios of the system, leaving open the question of whether the unexplored trajectories may contain the fatal bug, formal verification conducts an *exhaustive exploration* of all possible behaviors. Thus, when a design is pronounced correct by a formal verification method, it implies that all behaviors have been explored, and the questions of adequate coverage or a missed behavior become irrelevant.

Several approaches to formal verification have been proposed over the years. This book concentrates on the method of *model checking* by which a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that traverse through them.

Compared to other approaches, the *model checking* method enjoys two remarkable advantages:

- It is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. Anyone who can run simulations of a design is fully qualified and capable of model-checking the same design. In the context

of currently practiced techniques, model checking can be viewed as the ultimately superior simulation tool.

- When the design fails to satisfy a desired property, the process of model checking always produces a *counterexample* that demonstrates a behavior which falsifies the property. This faulty trace provides a priceless insight to understanding the real reason for the failure as well as important clues for fixing the problem.

These two significant advantages and the advent of *symbolic model checking*, which allows exhaustive implicit enumeration of an astronomic number of states, completely revolutionized the field of formal verification and transformed it from a purely academic discipline into a viable practical technique that can potentially be integrated as an additional valuable method for design validation within many industrial development processes.

An ample evidence of the wide industrial recognition of the great practical potential of model checking is provided by the large number of researchers and developers who work on the development of in-house model checkers and their applications within most of the advanced semiconductor and processor manufacturers big companies.

We are very fortunate that finally a definitive textbook on the principles and methods of model checking is available, written by authors who helped conceive the idea of model checking in the first place, and followed it through with impressive ingenuity and perseverance until it became the amazing success story it is.

I am fully confident that this textbook will provide an excellent reference and introduction to many readers, students, and practitioners who are interested in the exciting promising discipline of formal verification and its implementation by model checking.

Amir Pnueli

Preface

Finite-state concurrent systems arise naturally in several areas of computer science, particularly in the design of digital circuits and communication protocols. Logical errors found late in the design phase of these systems are an extremely important problem for both circuit designers and programmers. Such errors can delay getting a new product on the market or cause the failure of some critical device that is already in use. The most widely used verification technique is based on extensive testing or simulation and can easily miss significant errors when the number of possible states of the circuit or protocol is very large. Although there has been considerable research on the use of theorem provers, term rewriting systems, and proof checkers for verification, these techniques are time-consuming and often require a great deal of manual intervention. In the 1980s, an alternative verification technique called *temporal logic model checking* was developed independently by Clarke and Emerson [61] in the United States and by Quielle and Sifakis [219] in France. In this approach specifications are expressed in a propositional temporal logic, and circuit designs and protocols are modeled as state-transition systems. An efficient search procedure is used to determine if the specification is true of the transition system. In other words, the transition system is *checked* to see whether it is a *model* of the specification.

Model checking has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols. The most important is that the procedure is completely automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The procedure is also quite fast, and usually produces an answer in a matter of minutes. Partial specifications can be checked, so it is unnecessary to specify completely the system before useful information can be obtained regarding its correctness. When a specification is not satisfied, other formulas—not part of the original specification—can be checked in order to locate the source of the error. Finally, the logic used for specifications can directly express many of the properties that are needed for reasoning about concurrent systems.

The main disadvantage of model checking is the *state explosion* that can occur if the system being verified has many components that can make transitions in parallel. In this case the number of global system states may grow exponentially with the number of processes. Because of this problem, many researchers in formal verification predicted that model checking would never be practical for large systems. However, in the late 1980s the size of the transition systems that could be verified by model-checking techniques increased dramatically.

Much of the increase has been due to the use of *binary decision diagrams*, a data structure for representing boolean functions. The new data structure makes it possible to obtain concise representations for transition systems and to manipulate them quickly. This method is particularly useful for synchronous circuits. In the case of asynchronous protocols, it is possible to decrease the size of the state space by using the *partial order reduction*. This method is based on the following observation: computations that differ in the ordering of independently executed events are usually indistinguishable by the specification and can be considered equivalent. Thus, it is only necessary to check a reduced state space, which contains at least one representative computation for each such equivalence class.

As a result of these techniques and others, which will be described later in this book, model checking is now becoming widely used in industry as a practical verification technique. In fact, several companies are beginning to market model-checking tools.

We intend for this book to be used both as an introduction to model checking, and as a reference for researchers. We have tried to make it self-contained and as complete as possible. However, research in this area is moving so rapidly that it has been impossible to keep up with many new and exciting developments. Some parts of the book are more technical than others and can be safely skipped on the first reading. We have tried to indicate these sections, which are primarily intended for practitioners and researchers. We sincerely hope that this book will stimulate further research in model checking.

Finally, the authors would like to thank the people who have helped make this book possible. First and foremost we would like to express our appreciation to David Long, whose help has made this book a reality. We would also like to thank those people who read and commented on earlier drafts of this book: Eric Allen, Ilan Beer, Armin Biere, Sergey Berezin, Sergio Campos, Ching-Tsun Chou, Allen Emerson, Kousha Etessami, Nissim Francez, Masahiro Fujita, Yair Harel, Wolfgang Heinle, Hiromi Hiraishi, Neil Immerman, Somesh Jha, Irit Katriel, Shmuel Katz, Bob Kurshan, Kim G. Larsen, Yuan Lu, Jan Maluszynski, Will Marrero, Marius Minea, Bud Mishra, Ulf Nilsson, Wojciech Penczek, Amir Pnueli, Toshio Sekiguchi, Subash Shankar, Zeev Shtadler, Prasad Sistla, Frank Stomp, Wolfgang Thomas, Moshe Vardi, Dong Wang, Pierre Wolper, Bwolen Yang, Husnu Yenigün, Yunshan Zhu. We apologize if we have accidentally omitted other people who helped with the book. Edmund Clarke wishes to thank Michael Shostak for his encouragement during the period in which this book was written. Doron Peled would like to thank Marta Habermann, who provided a home away from home when he spent the spring semester of 1998 at CMU.

Model Checking

1 Introduction

Model checking is an automatic technique for verifying finite state concurrent systems. It has a number of advantages over traditional approaches to this problem that are based on simulation, testing, and deductive reasoning. The method has been used successfully in practice to verify complex sequential circuit designs and communication protocols. The main challenge in model checking is dealing with the *state space explosion* problem. This problem occurs in systems with many components that can interact with each other or systems that have data structures that can assume many different values (for example, the data path of a circuit). In such cases the number of global states can be enormous. During the past ten years considerable progress has been made in dealing with this problem. In this chapter we compare model checking with other formal methods for verifying hardware and software designs. We describe how model checking is used to verify complex system designs. We also trace the development of different model checking algorithms and discuss various approaches that have been proposed for dealing with the state explosion problem.

1.1 The Need for Formal Methods

Today, hardware and software systems are widely used in applications where failure is unacceptable: electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments, and other examples too numerous to list. We frequently read of incidents where some failure is caused by an error in a hardware or software system. A recent example of such a failure is the Ariane 5 rocket, which exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident found that it was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch, an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the computer to fail. The same error also caused the backup computer to fail. As a result incorrect attitude data was transmitted to the on-board computer, which caused the destruction of the rocket. The team investigating the failure suggested that several measures be taken in order to prevent similar incidents in the future, including the verification of the Ariane 5 software.

Clearly, the need for reliable hardware and software systems is critical. As the involvement of such systems in our lives increases, so too does the burden for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. We are very much dependent on such systems for continuous

operation; in fact, in some cases, devices are less safe when they are shut down. Even when failure is not life-threatening, the consequences of having to replace critical code or circuitry can be economically devastating.

Because of the success of the *Internet* and *embedded systems* in automobiles, airplanes, and other safety critical systems, we are likely to become even more dependent on the proper functioning of computing devices in the future. In fact, the pace of change will likely accelerate in coming years. Because of this rapid growth in technology, it will become even more important to develop methods that increase our confidence in the correctness of such systems.

1.2 Hardware and Software Verification

The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. Simulation and testing [202] both involve making experiments before deploying the system in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In the case of circuits, simulation is performed on the design of the circuit, whereas testing is performed on the circuit itself. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. For software, simulation and testing usually involve providing certain inputs and observing the corresponding outputs. These methods can be a cost-efficient way to find many errors. However, checking *all* of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible.

The term *deductive verification* normally refers to the use of axioms and proof rules to prove the correctness of systems. In early research on deductive verification, the main focus was on guaranteeing the correctness of critical systems. It was assumed that the importance of their correct behavior was so great that the developer or a verification expert (usually a mathematician or a logician) would spend whatever time was required for verifying the system. Initially, such proofs were constructed entirely by hand. Eventually, researchers realized that software tools could be developed to enforce the correct use of axioms and proof rules. Such tools can also apply a systematic search to suggest various ways to progress from the current stage of the proof.

The importance of deductive verification is widely recognized by computer scientists. It has significantly influenced the area of software development (for example, the notion of an *invariant* originated in research on deductive verification). However, deductive verification is a time-consuming process that can be performed only by experts

who are educated in logical reasoning and have considerable experience. The proof of a single protocol or circuit can last days or months. Consequently, use of deductive verification is rare. It is applied primarily to highly sensitive systems such as *security protocols*, where enough resources need to be invested to guarantee their safe usage.

It is important to realize that some mathematical tasks cannot be performed by an algorithm. The theory of *computability* [142] provides limitations on what can be decided by an algorithm. In particular, it shows that there cannot be an algorithm that decides whether an arbitrary computer program (written in some programming language like C or Pascal) terminates. This immediately limits what can be verified automatically. In particular, correct termination of programs cannot be verified automatically in general. Thus, most proof systems cannot be completely automated.

An advantage of deductive verification is that it can be used for reasoning about infinite state systems. This task can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof.

Model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be performed automatically. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. Given sufficient resources, the procedure will always *terminate* with a **yes/no** answer. Moreover, it can be implemented by algorithms with reasonable efficiency, which can be run on moderate-sized machines (but usually not on an average desktop computer).

Although the restriction to finite state systems may seem to be a major disadvantage, model checking is applicable to several very important classes of systems. Hardware controllers are finite state systems, and so are many communication protocols. In some cases, systems that are not finite state may be verified using model checking in combination with various abstraction and induction principles. Finally, in many cases errors can be found by restricting unbounded data structures to specific instances that are finite state. For example, programs with unbounded message queues can be debugged by restricting the size of the queues to a small number like two or three.

Because model-checking can be performed automatically, it is preferable to deductive verification, whenever it can be applied. However, there will always be some critical applications in which theorem proving is necessary for complete verification. An exciting new research direction [220] attempts to integrate deductive verification and model checking, so that the finite state parts of a complex system can be verified automatically.

1.3 The Process of Model Checking

Applying model checking to a design consists of several tasks, each of which will be discussed in detail later in this book.

Modeling The first task is to convert a design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task. In other cases, owing to limitations on time and memory, the modeling of a design may require the use of abstraction to eliminate irrelevant or unimportant details.

Specification Before verification, it is necessary to state the properties that the design must satisfy. The specification is usually given in some logical formalism. For hardware and software systems, it is common to use *temporal logic*, which can assert how the behavior of the system evolves over time.

An important issue in specification is *completeness*. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

Verification Ideally the verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

An error trace can also result from incorrect modeling of the system or from an incorrect specification (often called a *false negative*). The error trace can also be useful in identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model (e.g., by using additional abstractions).

1.4 Temporal Logic and Model Checking

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. They were originally developed by philosophers for investigating the way that time is used in natural language arguments [145]. Although a number of different temporal logics have been

studied, most have an operator like $\mathbf{G}\, f$ that is true in the present if f is always true in the future (*i.e.*, if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $\mathbf{G}(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. In this book the meaning of a temporal logic formula will always be determined with respect to a labeled state-transition graph; for historical reasons such structures are called *Kripke structures* [145].

Several researchers, including Burstall [48], Kröger [158] and Pnueli [216], have proposed using temporal logic for reasoning about computer programs. However, Pnueli [216] was the first to use temporal logic for reasoning about concurrency. His approach involved proving properties of the program under consideration from a set of axioms that described the behavior of the individual statements in the program. The method was extended to sequential circuits by Bochmann [25] and Malachi and Owicki [184]. Since proofs were constructed by hand, the technique was often difficult to use in practice.

The introduction of temporal-logic model checking algorithms by Clarke and Emerson [61, 103] in the early 1980s allowed this type of reasoning to be automated. Because checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently. The algorithm developed by Clarke and Emerson for the branching-time logic CTL was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how *fairness* [120] could be handled without changing the complexity of the algorithm. This was an important step in that the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [219] gave a model checking algorithm for a subset of CTL, but they did not analyze its complexity. Later Clarke, Emerson, and Sistla [63] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph. The algorithm was implemented in the EMC model checker, which was widely distributed and used to check a number of network protocols and sequential circuits [28, 29, 30, 31, 63, 98, 197]. Early model checking systems were able to check state transition graphs with between 10^4 and 10^5 states at a rate of about 100 states per second for typical formulas. In spite of these limitations, model checking systems were used successfully to find previously unknown errors in several published circuit designs.

Sistla and Clarke [232, 233] analyzed the model checking problem for a variety of temporal logics and showed, in particular, that for linear temporal logic (LTL) the problem was PSPACE-complete. Pnueli and Lichtenstein [173] reanalyzed the complexity of checking

linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. The same year, Fujita [119] implemented a tableau based verification system for LTL formulas and showed how it could be used for hardware verification.

CTL* is a very expressive logic that combines both branching-time and linear-time operators. The model checking problem for this logic was first considered in a paper by Clarke, Emerson, and Sistla [62], where it was shown to be PSPACE-complete, establishing that it is in the same general complexity class as the model checking problem for LTL. This result can be sharpened to show that CTL* and LTL model checking are of the same algorithmic complexity (up to a constant factor) in both the size of the state graph and the size of the formula. Thus, for purposes of model checking, there is no practical complexity advantage to restricting oneself to a linear temporal logic [106].

Alternative techniques for verifying concurrent systems have been proposed by a number of other researchers. Many of these approaches use automata for specifications as well as for implementations. The implementation is checked to see whether its behavior conforms to that of the specification. Because the same type of model is used for both implementation and specification, an implementation at one level can also be used as a specification for the next level of refinement. The use of language containment is implicit in the work of Kurshan [1], which ultimately resulted in the development of a powerful verifier called COSPAN [132, 133, 162]. Vardi and Wolper [245] first proposed the use of ω-automata (automata over infinite words) for automated verification. They showed how the linear temporal logic model checking problem could be formulated in terms of language containment between ω-automata. Other notions of conformance between the automata have also been considered, including observational equivalence [77, 196, 224], and various refinement relations [77, 195, 223].

1.5 Symbolic Algorithms

In the original implementation of the model checking algorithm, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts however, the number of states in the global state transition graph was too large to handle. In the fall of 1987, McMillan [46, 191], then a graduate student at Carnegie Mellon University, realized that by using a symbolic representation for the state transition graphs, much larger systems could be

verified. The new symbolic representation was based on Bryant's *ordered binary decision diagrams* (OBDDs) [34]. OBDDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states—many orders of magnitude larger than could be handled by the explicit-state algorithms. By using the original CTL model checking algorithm [61] of Clarke and Emerson with the new representation for state transition graphs, it became possible to verify some examples that had more than 10^{20} states [46, 191]. Since then, various refinements of the OBDD-based techniques by other researchers have pushed the state count up to more than 10^{120} [43, 44].

The implicit representation is quite natural for modeling sequential circuits and protocols. Each state is encoded by an assignment of boolean values to the set of state variables associated with the circuit or protocol. The transition relation can therefore be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is then represented by a binary decision diagram. The model checking algorithm is based on computing fixpoints of *predicate transformers* that are obtained from the transition relation. The fixpoints are sets of states that represent various temporal properties of the concurrent system. In the new implementations, both the predicate transformers and the fixpoints are represented with OBDDs. Thus, it is possible to avoid explicitly constructing the state graph of the concurrent system.

The model checking system that McMillan developed as part of his doctoral dissertation thesis is called SMV [191]. It is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented as an OBDD from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. If the transition system does not satisfy some specification, the verifier will produce an execution trace that shows why the specification is false. The SMV system has been widely distributed, and a large number of examples have now been verified with it. These examples provide convincing evidence that SMV can be used to debug real industrial designs.

An impressive example that illustrates the power of symbolic model checking is the verification of the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991). Although development of the Futurebus+ cache coherence protocol began in 1988, all previous attempts to validate the protocol were based entirely on informal techniques. In the summer of 1992 researchers at Carnegie Mellon [66, 179]

constructed a precise model of the protocol in SMV language and then used SMV to show that the resulting transition system satisfied a formal specification of cache coherence. They were able to find a number of previously undetected errors and potential errors in the design of the protocol. This appears to be the first time that an automatic verification tool has been used to find errors in an IEEE standard.

One of the best indications of the power of the symbolic verification methods comes from studying how the CPU time required for verification grows asymptotically with larger and larger instances of the circuit or protocol. In many of the examples that have been considered by a variety of groups, this growth rate is a small polynomial in the number of components of the circuit [18, 43, 44].

A number of other researchers have independently discovered that OBDDs can be used to represent large state-transition systems. Coudert, Berthet, and Madre [81] have developed an algorithm for showing equivalence between two deterministic finite-state automata by performing a breadth first search of the state space of the product automata. They use OBDDs to represent the transition functions of the two automata in their algorithm. Similar algorithms have been developed by Pixley [213, 214, 215]. In addition, several groups including Bose and Fisher [26], Pixley [213], and Coudert, Madre, and Berthet [82] have experimented with model checking algorithms that use OBDDs.

In related work Bryant, Seger and Beatty [18, 37] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic. Specifications consist of precondition–postcondition pairs expressed in the logic. The precondition is used to restrict inputs and initial states of the circuit; the postcondition gives the property that the user wishes to check. Formulas in the logic have the form

$$p_0 \wedge \mathbf{X} p_1 \wedge \mathbf{X}^2 p_2 \wedge \cdots \wedge \mathbf{X}^{n-1} p_{n-1} \wedge \mathbf{X}^n p_n.$$

The syntax of the formulas is highly restricted compared to most other temporal logics used for specifying programs and circuits. In particular, the only logical operator that is allowed is conjunction, and the only temporal operator is *next time* (**X**). By limiting the class of formulas that can be handled, it is possible to check certain properties very efficiently.

1.6 Partial Order Reduction

Verifying software causes some problems for model checking. Software tends to be less structured than hardware. In addition, concurrent software is usually *asynchronous*, that is, most of the activities taken by different processes are performed independently, without a global synchronizing clock. For these reasons, the state explosion phenomenon is a particularly serious problem for software. Consequently, model checking has been used less

frequently for software verification than for hardware verification. Recently, considerable progress has been made on the state explosion problem for software. The most successful techniques for dealing with this problem are based on the *partial order reduction* [126, 209, 244]. These techniques exploit the independence of concurrently executed events. Two events are *independent* of each other when executing them in either order results in the same global state.

A common model for representing concurrent software is the *interleaving model*, in which all of the events in a single execution are arranged in a linear order called an *interleaving sequence*. Concurrently executed events appear arbitrarily ordered with respect to one another. Most logics for specifying properties of concurrent systems can distinguish between interleaving sequences in which two independent events are executed in different orders. Because of this, all possible interleavings of such events are normally considered. This can result in an extremely large state space.

The partial order reduction techniques make it possible to decrease the number of interleaving sequences that must be considered. As a result, the number of states that are needed for model checking is reduced. When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them. These methods are related to the *partial order model of program execution*. According to this model, concurrently executed events are not ordered. Each partially ordered execution can correspond to multiple interleaving sequences. If it is impossible to distinguish between such sequences, it is sufficient to select one interleaving sequence for each partial ordering of events.

The idea of reducing the state space by selecting only a subset of the ways one can interleave independently executed transitions has been studied by many researchers. One of the first researchers to propose such a reduction technique was Overman [205]. However, he only considered a restricted model of concurrency that did not include looping and nondeterministic choice. The proof system of Katz and Peled [153] suggests using an equivalence relation between interleaving sequences that correspond to the same partially ordered execution. Their system includes proof rules for reasoning about a selection of interleaved sequences rather than all of them. Model checking algorithms that incorporate the partial order reduction are described in several different papers. The *stubborn sets* of Valmari [244], the *persistent sets* of Godefroid [125] and the *ample sets* of Peled [209] differ on the actual details, but contain many similar ideas. In this book we will describe the ample set method. Other methods that exploit similar observations about the relation between the partial and total order models of execution are McMillan's *unfolding technique* [190] and Godefroid's *sleep sets* [125].

1.7 Other Approaches to the State Explosion Problem

Although symbolic representations and the partial order reduction have greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. Four such techniques are compositional reasoning, abstraction, symmetry, and induction.

The first technique exploits the *modular structure* of complex circuits and protocols [72, 128, 129, 150, 151, 168, 218, 230]. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties, using only the part of the system that it describes. If it is possible to show that the system satisfies each local property, and if the conjunction of the local properties implies the overall specification, then the complete system must satisfy this specification as well.

When this naive form of compositional reasoning is not feasible because of mutual dependencies between the components, a more complex strategy is necessary. In such cases, when verifying a property of one component we must make assumptions about the behavior of the other components. The assumptions must later be discharged when the correctness of the other components is established. This strategy is called *assume-guarantee reasoning* [129, 150, 151, 198, 218].

The second technique involves using *abstraction*. This technique appears to be essential for reasoning about reactive systems that involve data paths. Traditionally, finite state verification methods have been used mainly for control-oriented systems. The symbolic methods make it possible to handle some systems that involve nontrivial data manipulation, but the complexity of verification is often high. The use of abstraction is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. For example, in verifying the addition operation of a microprocessor, we might require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations *abstraction* can be used to reduce the complexity of model checking [20, 69, 90, 91, 160, 248]. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level.

Symmetry can also be used to reduce the state explosion problem [64, 111, 143, 148]. Finite state concurrent systems frequently contain replicated components. For example,

a large number of protocols involve a network of identical processes communicating in some fashion. Hardware devices contain parts such as memories and register files that have many replicated elements. These facts can be used to obtain reduced models for the system. Having symmetry in a system implies the existence of a nontrivial permutation group that preserves the state transition graph Such a group can be used to define an equivalence relation on the state space of the system and to reduce the state space The reduced model can be used to simplify the verification of properties of the original model expressed by a temporal logic formula.

Induction involves reasoning automatically about entire families of finite-state systems [33, 67, 155, 165, 187, 229, 250]. Such families arise frequently in the design of reactive systems in both hardware and software. Typically, circuit and protocol designs are parameterized, that is, they define an infinite family of systems. For example, a circuit designed to add two integers has the width of the integers n as a parameter; a bus protocol may be designed to accommodate an arbitrary number of processors, and a mutual exclusion protocol can be given for a parameterized number of processes. We would like to be able to check that every system in a given family satisfies some temporal logic property. In general the problem is undecidable [12, 237], but in many interesting cases, it is possible to provide a form of *invariant process* that represents the behavior of an arbitrary member of the family Using this invariant, we can then check the property for all of the members of the family at once An inductive argument is used to verify that the invariant is an appropriate representative

2 Modeling Systems

The first step in verifying the correctness of a system is specifying the properties that the system should have. For example, we may want to show that some concurrent program never deadlocks. Once we know which properties are important, the second step is to construct a *formal model* for the system. In order to be suitable for verification, the model should capture those properties that must be considered to establish correctness. On the other hand, it should abstract away those details that do not effect the correctness of the checked properties but make verification more complicated. For example, when modeling digital circuits, it is useful to reason in terms of gates and boolean values, rather than actual voltage levels. Likewise, when reasoning about a communication protocol we may want to focus on the exchange of messages and ignore the actual contents of the messages.

In this book, we will be primarily concerned with *reactive systems* [186] and their behavior over time. Such systems may need to interact with their environment frequently and often do not terminate. Therefore, they cannot adequately be modeled by their input-output behavior. The first feature of a reactive system that we want to capture is its *state*. A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time. We also need to know how the state of the system changes as the result of some action of the system. We can describe the change by giving the state before the action occurs and the state after the action occurs. Such a pair of states determines a *transition* of the system. The computations of a reactive system can be defined in terms of its transitions. A *computation* is an infinite sequence of states where each state is obtained from the previous state by some transition.

We use a type of state transition graph called a *Kripke structure* to capture this intuition about the behavior of reactive systems. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state. Paths in a Kripke structure model computations of the system. Although these models are very simple, they are sufficiently expressive to capture those aspects of temporal behavior that are most important for reasoning about reactive systems.

Concurrent systems are usually given by the text of a program or by the diagram for a circuit. There are many different types of concurrent systems (synchronous and asynchronous circuits, programs with shared variables, programs that communicate by message passing, and so on). Because of this diversity we need a unifying formalism that can represent a concurrent system of any type. We will use formulas of first order logic for this purpose. Given a formula that represents a concurrent system, it is straightforward to extract the Kripke structure that models the system.

In the following sections we formally define Kripke structures. We show how to extract such structures from first order formulas that represent concurrent systems. Finally, we demonstrate how different programming constructs can be represented in terms of first order formulas.

2.1 Modeling Concurrent Systems

Let AP be a set of atomic propositions. A *Kripke structure* M over AP is a four tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
4. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Sometimes we will not be concerned with the set of initial states S_0. In such cases, we will omit this set of states from the definition. A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

2.1.1 First Order Representations

We only assume a basic knowledge of first order logic. The reader should be familiar with the logical connectives (*and* $\wedge$, *or* $\vee$, *not* $\neg$, *implies* $\rightarrow$, and so on) and should know how universal ($\forall$) and existential ($\exists$) quantification work.

We use interpreted first order formulas to describe concurrent systems. Thus, the predicate and function symbols that occur in such formulas will have a predefined meaning. Usually, this meaning will be clear from the context. Let $V = \{v_1, \ldots, v_n\}$ be the set of system variables. We assume that the variables in V range over a finite set D (sometimes called the *domain* or *universe* of the interpretation). A *valuation* for V is a function that associates a value in D with each variable v in V.

A *state* of a concurrent system can be described by giving values for all of the elements in V. In other words, a state is just a valuation $s : V \rightarrow D$ for the set of variables in V. Given a valuation, we can write a formula that is true for exactly that valuation. For example, given $V = \{v_1, v_2, v_3\}$ and the valuation $\langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5\rangle$, we derive the formula $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$. In general, a formula may be true for many valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe certain sets of states by first order formulas. In particular, the *initial states* of the system can be described by a first order formula $\mathcal{S}_0$ over the variables in V.

In addition to representing sets of states, we must be able to represent sets of transitions between states. To do this, we extend the idea used above. This time, we use a formula to represent a set of ordered pairs of states. We cannot do this using just a single copy of the

system variables V, so we create a second set of variables V'. We think of the variables in V as *present state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable in V', which we denote by v'. A valuation for the variables in V and V' can be viewed as designating an ordered pair of states or a transition, and we can represent sets of these valuations using formulas as above. We refer to a set of pairs of states as a *transition relation*. If R is a transition relation, then we write $\mathcal{R}(V, V')$ to denote a formula that represents it.

In order to write specifications that describe properties of concurrent systems we need to define a set of atomic propositions AP. Atomic propositions will typically have the form $v = d$ where $v \in V$ and $d \in D$. A proposition $v = d$ will be true in a state s if $s(v) = d$. When v is a variable over the boolean domain $\{True, False\}$, it is not necessary to include both $v = True$ and $v = False$ in AP. We will write v to indicate that $s(v) = True$ and $\neg v$ to indicate that $s(v) = False$.

We now show how to derive a Kripke structure $M = (S, S_0, R, L)$ from the first order formulas $\mathcal{S}_0$ and $\mathcal{R}$ that represent the concurrent system.

- The set of states S is the set of all valuations for V.
- The set of initial states S_0 is the set of all valuations s_0 for V that satisfy the formula $\mathcal{S}_0$.
- Let s and s' be two states, then $R(s, s')$ holds if $\mathcal{R}$ evaluates to *True* when each $v \in V$ is assigned the value $s(v)$ and each $v' \in V'$ is assigned the value $s'(v)$.
- The labeling function $L : S \rightarrow 2^{AP}$ is defined so that $L(s)$ is the subset of all atomic propositions true in s. If v is a variable over the boolean domain, then $v \in L(s)$ indicates that $s(v) = True$, and $v \notin L(s)$ indicates that $s(v) = False$.

Because we require that the transition relation of a Kripke structure is always total, we must extend the relation R if some state s has no successor. In this case, we modify R so that $R(s, s)$ holds.

To illustrate the notions defined in this section we consider a simple system with variables x and y that range over $D = \{0, 1\}$. Thus, a valuation for the variables x and y is just a pair $(d_1, d_2) \in D \times D$ where d_1 is the value for x and d_2 is the value for y. The system consists of one transition

$$x := (x + y) \bmod 2,$$

which starts from the state in which $x = 1$ and $y = 1$. This system will be described by two first order formulas. The set of initial states of the system is represented by

$$\mathcal{S}_0(x, y) \equiv x = 1 \wedge y = 1,$$

and the set of transitions is represented by

$\mathcal{R}(x, y, x', y') \equiv x' = (x + y) \; mod \; 2 \; \wedge \; y' = y.$

The Kripke structure $M = (S, S_0, R, L)$ extracted from these formulas is:

- $S = D \times D$.
- $S_0 = \{(1, 1)\}$.
- $R = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\}$.
- $L((1, 1)) = \{x = 1, y = 1\}$, $L((0, 1)) = \{x = 0, y = 1\}$, $L((1, 0)) = \{x = 1, y = 0\}$, and $L((0, 0)) = \{x = 0, y = 0\}$.

The only path in the Kripke structure that starts in an initial state is (1,1) (0,1) (1,1) (0,1) . . . . This path is the only computation of the system.

2.1.2 Granularity of Transitions

A critical issue in modeling concurrent systems is determining the granularity of the transitions. It is important to obtain transitions that are *atomic* in the sense that no observable state of the system can result from executing part of a transition. A common mistake is to define transitions that are too coarse. In this case, the Kripke structure may not include some states that are observable. As a result, verification techniques such as model checking may fail to find important errors. A problem can also arise when the granularity is too fine. In this case transitions can interact to create new states that are not reachable in the actual system. As a result, model checking may find spurious errors that will never occur in practice.

For an example, consider a system with two variables x and y and two transitions α and β that can be executed concurrently.

α: $x := x + y$ and

β: $y := y + x$

with the initial state $x = 1 \wedge y = 2$. Also consider a finer grained implementation of the same transitions. This implementation uses the assembly language instructions for loading, adding, and storing between a memory address and a register:

α_0: load R_1, x	β_0: load R_2, y
α_1: add R_1, y	β_1: add R_2, x
α_2: store R_1, x	β_2: store R_2, y

Executing α and then β results in the state $x = 3 \wedge y = 5$. When β is executed before α, we obtain $x = 4 \wedge y = 3$. If, on the other hand, the finer grained implementation is executed in the order $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$, the result is $x = 3 \wedge y = 3$.

Suppose that $x = 3 \wedge y = 3$ violates some desired property of the system. Further suppose that the system is implemented using the transitions α and β. Then, it is impossible to have $x = 3$ and $y = 3$ at the same time. However, if we model the system with the finer grained transitions $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2, we may erroneously conclude that the system is incorrect. Next, suppose that the system is implemented using $\alpha_0, \alpha_1, \alpha_2, \beta_0, \beta_1$, and β_2. In this case it is possible to reach a state in which both $x = 3$ and $y = 3$. If we now model the system with α and β, we will erroneously conclude that the system is correct.

Extracting a first order representation from the text of a program or a diagram of a circuit can be viewed as a compilation task. This task must take into account granularity considerations like the one described above. In the next section, we will discuss in greater detail how the compilation is performed.

2.2 Concurrent Systems

A concurrent system consists of a set of components that execute together. Normally the components have some means of communicating with each other. The mode of execution and the mode of communication may differ from one system to another. We will consider two modes of execution: *Asynchronous* or *interleaved execution*, in which only one component makes a step at a time, and *synchronous execution*, in which all of the components make a step at the same time. We will also distinguish three modes of communication. Components can either communicate by changing the values of *shared variables* or by *exchanging messages* using queues or some type of handshaking protocol. Because modeling is not the main concern of this book, we will only discuss communication by shared variables in this chapter.

In the following sections we describe some important types of concurrent systems and show how they can be represented in terms of first order formulas. From these formulas we can derive Kripke structures for the systems, as shown in Section 2.1.1.

2.2.1 Digital Circuits

In this section, we show how to describe circuits by formulas. For simplicity, we assume that each *state holding element* of a circuit can have the value 0 or 1. Let V be the set of state holding elements of a circuit. For a synchronous circuit, the set V typically consists of the outputs of all the registers in the circuit together with the primary inputs. For asynchronous circuits, all wires in the circuit are usually considered to be state holding elements. If we create a boolean variable for each element in V, then a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can write a boolean expression that is true for exactly that valuation. For example, given

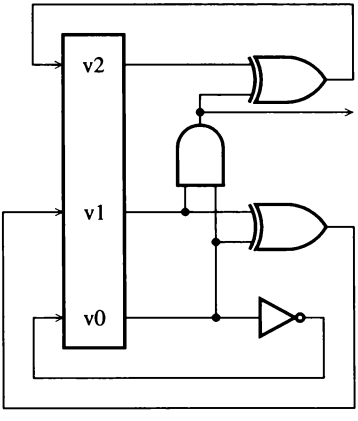


Figure 2.1
Synchronous modulo 8 counter.

$V = \{v_1, v_2\}$ and the valuation $\langle v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$, we derive the boolean formula $v_1 \wedge \neg v_2$. As before, we adopt the convention that a formula represents the set of *all* valuations that make it true. Thus, for describing circuits the full expressive power of first order logic is not needed; boolean formulas are sufficient. The boolean formulas $\mathcal{S}_0(V)$ and $\mathcal{R}(V, V')$ will represent the set of initial states and the transition relation of the circuit, respectively.

Synchronous Circuits

The operation of a synchronous circuit consists of a sequence of steps. In each step, the inputs to the circuit change and the circuit is allowed to stabilize. Then a clock pulse occurs, and the state-holding elements change.

The method for deriving the transition relation of a synchronous circuit can be illustrated using a small example. The circuit in Figure 2.1 is a modulo 8 counter. Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v_0', v_1', v_2'\}$ be another copy of the state variables. The transitions of the modulo 8 counter are given by

$$v_0' = \neg v_0$$

$$v_1' = v_0 \oplus v_1$$

$$v_2' = (v_0 \wedge v_1) \oplus v_2$$

where $\oplus$ is the *exclusive or* operator. The above equations can be used to define the relations

$$\mathcal{R}_0(V, V') \equiv (v_0' \Leftrightarrow \neg v_0)$$

$$\mathcal{R}_1(V, V') \equiv (v_1' \Leftrightarrow v_0 \oplus v_1)$$

$$\mathcal{R}_2(V, V') \equiv (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

which describe the constraints each v_i' must satisfy in a legal transition. Because all the changes occur at the same time, the constraints are combined by taking their conjunction to construct a formula for the transition relation:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \mathcal{R}_1(V, V') \wedge \mathcal{R}_2(V, V').$$

In the general case of a synchronous circuit with n state holding elements, we let $V = \{v_0, \cdots, v_{n-1}\}$ and $V' = \{v_0', \ldots, v_{n-1}'\}$. Analogous to the modulo 8 counter, for each state variable v_i' there is a boolean function f_i such that

$$v_i' = f_i(V).$$

These equations are used to define the relations

$$\mathcal{R}_i(V, V') \equiv (v_i' \Leftrightarrow f_i(V)).$$

Continuing the analogy with the modulo 8 counter, the conjunction of these formulas forms the transition relation

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \wedge \cdots \wedge \mathcal{R}_{n-1}(V, V').$$

Thus, the transition relation for a synchronous circuit can be expressed as the conjunction of the transition relations of the individual processes.

Asynchronous Circuits

The transition relation for an asynchronous circuit is most naturally expressed as a disjunction. To simplify the description of how the transition relations are obtained, we assume that all the components of the circuits have exactly one output and have no internal state variables. In this case, it is possible to describe each component by a function $f_i(V)$; given values for the present state variables v, the component drives its output to the value specified by $f_i(V)$. Extending the method to handle components with multiple outputs is straightforward.

Because the value of a component can change so rapidly, it is unlikely that two components will change at the same time. For this reason, it is customary to use an *interleaving semantics* in which exactly one component changes at a time. This results in a disjunction of the form:

$$\mathcal{R}(V, V') \equiv \mathcal{R}_0(V, V') \vee \cdots \vee \mathcal{R}_{n-1}(V, V'),$$

where

$$\mathcal{R}_i(V, V') \equiv \big(v_i' \Leftrightarrow f_i(V)\big) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

Note that some component may change repeatedly, without another component ever making a step. In practice, this is extremely unlikely. It is possible to augment the model with an additional *fairness* constraint that will disallow such behaviors. This topic will be discussed further in the next chapter.

To illustrate the difference between the synchronous and the asynchronous models, consider the following example. Let $V = \{v_0, v_1\}$, $v_0' = v_0 \oplus v_1$ and $v_1' = v_0 \oplus v_1$. Let s be a state with $v_0 = 1 \wedge v_1 = 1$. According to the synchronous model, the only successor of s is the state with $v_0 = 0 \wedge v_1 = 0$, since both assignments are executed simultaneously. According to the asynchronous model, the state s has two successors:

1. $v_0 = 0 \wedge v_1 = 1$ (the assignment to v_0 is taken first).
2. $v_0 = 1 \wedge v_1 = 0$ (the assignment to v_1 is taken first).

2.2.2 Programs

All of the programs we consider are asynchronous. We start by discussing sequential programs because concurrent programs are composed of sequential components. The approach that we use is similar to the approach used in the book by Manna and Pnueli [186]. For a more detailed treatment of these issues, we refer the reader to that book. A program consists of statements that are sequentially composed with each other. We describe a translation procedure $\mathcal{C}$ that takes the text of a sequential program P and transforms it into a first order formula $\mathcal{R}$ that represents the set of transitions of the program. Without loss of generality, we assume that each statement has a unique *entry point* and a unique *exit point*. The transition procedure is simplified significantly if each entry and exit point of a statement in the program is uniquely labeled. Thus, we define a labeling transformation that given an unlabeled program P results in a labeled program $P^{\mathcal{L}}$.

The labeling transformation defined below attaches a single label with the entry point of each statement in P, except for P itself. No two attached labels are identical. In sequential programs, the exit point of a statement is identical to the entry point of the following statement. Thus, it is sufficient to label entry points. If we also provide labels for the entry and the exit points of P, then we get a unique labeling of the entry and exit points of all statements of the program.

Since we do not restrict ourselves to a specific programming language, we define the labeling transformation for a number of statement types. It is easy to extend the definition to other statement types. Given a statement P, the *labeled statement* $P^{\mathcal{L}}$ is defined as follows:

- If P is not a composite statement (e.g., P is $x := e$, **skip**, **wait**, **lock**, **unlock**, etc.), then $P^{\mathcal{L}} = P$.
- If $P = P_1; P_2$ then $P^{\mathcal{L}} = P_1^{\mathcal{L}};\ l'' : P_2^{\mathcal{L}}$.
- If $P =$ **if** b **then** P_1 **else** P_2 **end if**, then $P^{\mathcal{L}} =$ **if** b **then** $l_1 : P_1^{\mathcal{L}}$ **else** $l_2 : P_2^{\mathcal{L}}$ **end if**.
- If $P =$ **while** b **do** P_1 **end while**, then $P^{\mathcal{L}} =$ **while** b **do** $l_1 : P_1^{\mathcal{L}}$ **end while**.

In the remainder of this section, we assume that P is a labeled statement and that the entry and exit points of P are labeled by m and m' respectively. Let pc be a special variable called the *program counter* that ranges over the set of program labels and an additional value $\perp$ called the *undefined value*. The undefined value is needed when concurrent programs are considered. In this case, $pc = \perp$ indicates that the program is not active.

Let V denote the set of program variables. Let V' be the set of primed variables v' for each $v \in V$, and let pc' be the primed variable for pc. Recall that the unprimed copy refers to the value of the variables before a transition, whereas the primed copy refers to the value after the transition. Because each transition typically changes only a small number of the program variables, we will use $same(Y)$ as an abbreviation for the formula

$$\bigwedge_{y \in Y} (y' = y).$$

We first give the formula that describes the set of initial states of the program P. Given some condition $pre(V)$ on the initial values of the variables of P,

$$\mathcal{S}_0(V, pc) \equiv pre(V) \wedge pc = m.$$

The translation procedure $\mathcal{C}$ depends on three parameters: the entry label l, the labeled statement P, and the exit label l'. The procedure is defined recursively with one rule for each statement type in the language. $\mathcal{C}(l,\ P,\ l')$ describes the set of transitions in P as a disjunction of all the transitions in the set. The disjunct for an individual transition determines the value of the boolean condition and the value of the program counter for which the transition may be executed. It is true whenever the transition is enabled and false otherwise.

- Assignment:

$$\mathcal{C}(l,\ v \leftarrow e,\ l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \setminus \{v\})$$

- Skip:

$$\mathcal{C}(l,\ skip,\ l') \equiv pc = l \wedge pc' = l' \wedge same(V)$$

- Sequential composition:

$$\mathcal{C}(l,\ P_1;\ l'' : P_2,\ l') \equiv \mathcal{C}(l,\ P_1,\ l'') \vee \mathcal{C}(l'',\ P_2,\ l')$$

The formula for the transitions of $P_1;\ l'' : P_2$ is a disjunction of the formulas for the transitions of P_1 and of P_2. Because of the intermediate label l'', statement P_2 will be executed after statement P_1.

- Conditional:

$\mathcal{C}(l,$ **if** b **then** $l_1 : P_1$ **else** $l_2 : P_2$ **end if**, $l')$ is the disjunction of the following four formulas:

- $pc = l \wedge pc' = l_1 \wedge b \wedge same(V)$
- $pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V)$
- $\mathcal{C}(l_1,\ P_1,\ l')$
- $\mathcal{C}(l_2,\ P_2,\ l')$

The first disjunct corresponds to the case where condition b is true. In this case, statement P_1 will be executed next. The second disjunct corresponds to the case where condition b is false. In this case, statement P_2 will be executed next. Both disjuncts describe transitions that involve only a change of the program counter. The third and fourth disjuncts are formulas for the transitions of P_1 and P_2, respectively. Note that l' is the exit point for both P_1 and P_2. The translation for the **if** statement can easily be extended to handle nondeterministic choice between several alternatives.

- While:

$\mathcal{C}(l,$ **while** b **do** $l_1 : P_1$ **end while**, $l')$ is the disjunction of the following three formulas:

- $pc = l \wedge pc' = l_1 \wedge b \wedge same(V)$
- $pc = l \wedge pc' = l' \wedge \neg b \wedge same(V)$
- $\mathcal{C}(l_1,\ P_1,\ l)$

The first disjunct corresponds to the case where condition b is true. In this case, statement P_1 will be executed next. The second disjunct corresponds to the case where condition b is false, in which case, the execution of the while statement terminates. The third disjunct is a formula for the set of transitions of P_1. Note that the exit point of P_1 is identical to the entry point of the while statement. Thus, if P_1 terminates the execution of the while statement will restart.

2.2.3 Concurrent Programs

A *concurrent program* consists of a set of processes that can be executed in parallel. A *process* is a sequential statement as described in the previous section. Concurrent programs in which processes do not interact by means of message passing or shared variables are

usually easy to analyze and will not be considered further. In this section, we will consider asynchronous programs in which exactly one process can make a transition at any time. We begin by introducing some terminology that will be used throughout the section. V_i is the set of variables that can be changed by process P_i. We do not require that these sets be disjoint. As before, V is the set of all program variables. The program counter of a process P_i is pc_i. PC is the set of all program counters.

A concurrent program P has the form

cobegin $P_1 \| \; P_2 \| \ldots \| \; P_n$ **coend**

where $P_1, \ldots, P_n$ are processes. The labeling transformation for sequential programs is extended so that a concurrent program can occur as a statement in a sequential program. The transformation attaches a label to the entry point and to the exit point of each process. Unlike exit points in sequential programs, no exit point of a concurrent process is identical to an entry point. As a result, the exit points of processes must be explicitly labeled. As before, we assume that no two labels are identical and that the entry and exit points of P are labeled m and m', respectively.

- If $P =$ **cobegin** $P_1 \| \; P_2 \| \ldots \| \; P_n$ **coend**, then
$P^{\mathcal{L}} =$ **cobegin** $l_1 : P_1^{\mathcal{L}} \; l_1' \| \; l_2 : P_2^{\mathcal{L}} \; l_2' \| \ldots \| \; l_n : P_n^{\mathcal{L}} \; l_n'$ **coend**.

The formula that describes the initial states of a concurrent program P is

$$\mathcal{S}_0(V, PC) \equiv pre(V) \wedge pc = m \wedge \boxed{\bigwedge_{i=1}^{n} (pc_i = \bot)},$$

where $pc_i = \bot$ indicates that process P_i has not been activated yet and therefore cannot be executed from the current state.

The translation procedure $\mathcal{C}$ is extended to concurrent programs as follows: $\mathcal{C}(l,$ **cobegin** $l_1 : P_1 \; l_1' \| \ldots \| \; l_n : P_n \; l_n'$ **coend**, $l')$ is the disjunction of three formulas:

- $pc = l \;\wedge\; pc_1' = l_1 \;\wedge\; \ldots \;\wedge\; pc_n' = l_n \;\wedge\; pc' = \bot$
- $pc = \bot \;\wedge\; pc_1 = l_1' \;\wedge\; \ldots \;\wedge\; pc_n = l_n' \;\wedge\; pc' = l' \;\wedge \bigwedge_{i=1}^{n} (pc_i' = \bot)$
- $\bigvee_{i=1}^{n} \big(\mathcal{C}(l_i, P_i, l_i') \;\wedge\; same(V \setminus V_i) \;\wedge\; same(PC \setminus \{ pc_i \})\big)$

The first disjunct describes the initialization of the concurrent processes. A transition is made from the entry point of the **cobegin** statement to the entry points of the individual processes. The second disjunct describes the termination of the concurrent program. A transition is made from the exit points of the processes to the exit of the **cobegin** statement. This transition will only be executed if all the processes terminate. The third disjunct

describes the interleaved execution of the concurrent processes. The formula for the transition relation of process P_i is conjuncted with $same\ (V \setminus V_i)\ \wedge same(PC \setminus \{\ pc_i\ \}))$. This guarantees that a transition in process P_i can only change variables in V_i. It also ensures that only one process can make a transition at any time.

Page 26

Shared Variables

Recall that V_i is the set of variables that may be changed by process P_i. Concurrent programs for which the sets V_i overlap are called *shared variable* programs. We show how to extend the translation procedure $\mathcal{C}$ to some commonly used *process synchronization* statements. Such statements are frequently needed to provide processes with exclusive access to shared variables. These statements are atomic and treated by the labeling transformation accordingly. Assume that the statement belongs to the text of process P_i.

- Wait: Because our primary interest is in finite state programs, we only describe how to implement this statement using *busy waiting*. In particular, we do not consider implementations that require complex data structures like process queues. The statement **wait**(b) repeatedly tests the value of the boolean variable b until it determines that b is true. When b becomes true, a transition is made to the next program point

  $\mathcal{C}(l,\ \mathbf{wait}(b),\ l')$ is a disjunction of the following two formulas:

- $(pc_i = l\ \wedge\ pc_i' = l\ \wedge\ \neg b\ \wedge\ same(V_i))$
- $(pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ b\ \wedge\ same(V_i))$

- Lock: The statement **lock**(v) is similar to the statement **wait**$(v = 0)$, except that when $v = 0$ is true the transition changes the value of v to 1. This statement is often used to guarantee *mutual exclusion* by preventing more than one process from entering its critical region.

  $\mathcal{C}(l,\ \mathbf{lock}(v),\ l')$ is a disjunction of the following two formulas: Page 26

- $(pc_i = l\ \wedge\ pc_i' = l\ \wedge\ v = 1\ \wedge\ same(V_i))$
- $(pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ v = 0\ \wedge\ v' = 1\ \wedge same(V_i \setminus \{v\}))$

- Unlock: The statement **unlock**(v) assigns the value 0 to the variable v. Typically, this statement enables some other process to enter its critical region.

$$\mathcal{C}(l,\ \mathbf{unlock}(v),\ l')\ \equiv\ pc_i = l\ \wedge\ pc_i' = l'\ \wedge\ v' = 0\ \wedge\ same(V_i \setminus \{v\})$$

2.3 Example of Program Translation

Consider a simple *mutual exclusion* program

$$P = m : \mathbf{cobegin}\ P_0 \parallel P_1\ \mathbf{coend}\ m'$$

with two processes P_0 and P_1, where

$P_0 ::$ $l_0 :$ **while** *True* **do**
$NC_0 :$ **wait**$(turn = 0)$;
$CR_0 :$ $turn := 1$;
end while;
l'_0

$P_1 ::$ $l_1 :$ **while** *True* **do**
$NC_1 :$ **wait**$(turn = 1)$;
$CR_1 :$ $turn := 0$;
end while;
l'_1

The program counter pc of the program P takes only three values: m, the label of the entry point of P; m', the label of the exit point of P; and $\perp$ the value of pc when P_1 and P_2 are active. Each process P_i has a program counter pc_i that ranges over the labels l_i, l'_i, NC_i, CR_i, and $\perp$. The two processes share a single variable $turn$. Thus, $V = V_0 = V_1 = \{turn\}$ and $PC = \{pc, pc_0, pc_1\}$. When the value of the program counter of a process P_i is CR_i, the process is in its *critical region*. Both processes are not allowed to be in their critical regions at the same time. When the value of the program counter is NC_i, the process is in its *noncritical region*. In this case it waits until $turn = i$ in order to gain exclusive entry into the critical region.

The initial states of P are described by the formula

$$\mathcal{S}_0(V, PC) \equiv pc = m \wedge pc_0 = \perp \wedge pc_1 = \perp.$$

Note that no restriction is imposed on the value of $turn$. Thus, it may initially be either 0 or 1. Applying the translation procedure $\mathcal{C}$ we obtain the formula for the transition relation of P, $\mathcal{R}(V, PC, V', PC')$, which is the disjunction of the following four formulas:

- $pc = m \wedge pc'_0 = l_0 \wedge pc'_1 = l_1 \wedge pc' = \perp$
- $pc_0 = l'_0 \wedge pc_1 = l'_1 \wedge pc' = m' \wedge pc'_0 = \perp \wedge pc'_1 = \perp$
- $\mathcal{C}(l_0, P_0, l'_0) \wedge same(V \setminus V_0) \wedge same(PC \setminus \{ pc_0 \})$, which is equivalent to

$$\mathcal{C}(l_0, P_0, l'_0) \wedge same(pc, pc_1)$$

- $\mathcal{C}(l_1, P_1, l'_1) \wedge same(V \setminus V_1) \wedge same(PC \setminus \{ pc_1 \})$, which is equivalent to

$$\mathcal{C}(l_1, P_1, l'_1) \wedge same(pc, pc_0)$$

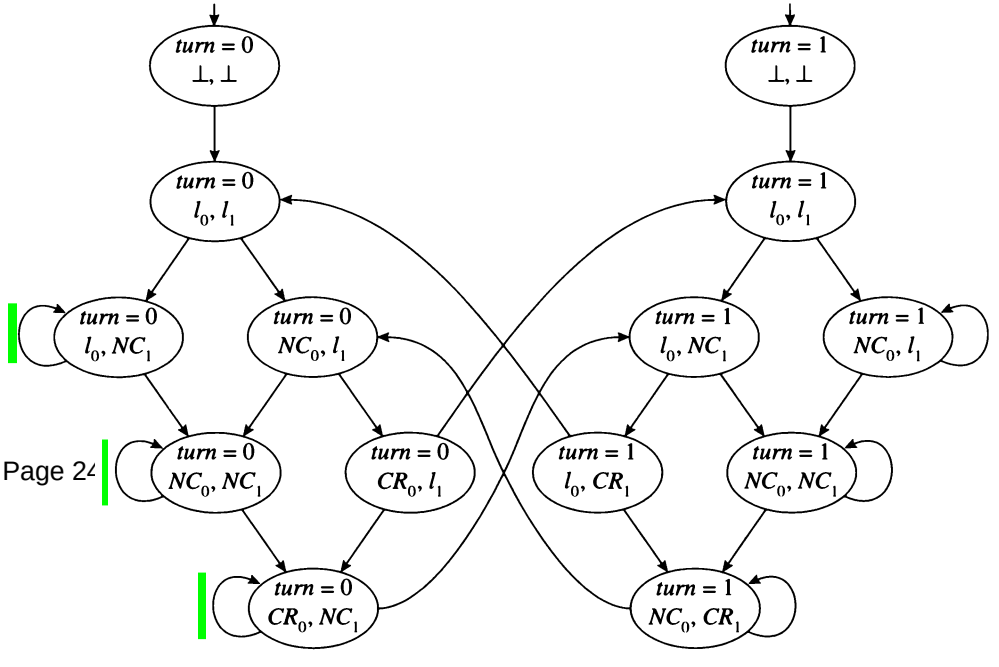


Figure 2.2
Reachable states of Kripke structure for mutual exclusion example.

For each process P_i, $\mathcal{C}(l_i, P_i, l_i')$ is the disjunction of:

- $pc_i = l_i \ \wedge \ pc_i' = NC_i \ \wedge \ True \ \wedge \ same(turn)$ Page 22
- $pc_i = NC_i \ \wedge \ pc_i' = CR_i \ \wedge \ turn = i \ \wedge \ same(turn)$
- $pc_i = CR_i \ \wedge \ pc_i' = l_i \ \wedge \ turn' = (i+1) \ mod \ 2$ Page 24
- $pc_i = NC_i \ \wedge \ pc_i' = NC_i \ \wedge \ turn \neq i \ \wedge \ same(turn)$
- $pc_i = l_i \ \wedge \ pc_i' = l_i' \ \wedge \ False \ \wedge \ same(turn)$ Page 22

The Kripke structure in Figure 2.2 is derived from the formulas $\mathcal{S}_0$ and $\mathcal{R}$ as described in Section 2.1.1. By examining the state space of the Kripke structure, it is easy to see that the processes will never be in their critical regions at the same time. Thus, the program guarantees the required mutual exclusion property. However, this program fails to guarantee *absence of starvation*, since one of the processes may continuously try to enter its critical region without ever being able to do so, while the other process stays in its critical region forever. Later, we will see how to formulate and model check such properties.

3 Temporal Logics

In this chapter we describe a logic for specifying properties of the state transition systems or Kripke structures introduced in Section 2.1. The logic uses atomic propositions and boolean connectives such as conjunction, disjunction, and negation to build up complicated expressions describing properties of states. In *reactive* systems, we are also interested in describing the transitions between states. This is important because such systems interact with and continually respond to their environment. Traditional software verification methodologies, such as those due to Floyd [114] and Hoare [136], deal with the input-output semantics of programs. The internal details of how the computation is carried out are not reflected in the properties that can be specified and proved; only the input at the start of execution and the output at termination are described. In contrast, for reactive systems, the computation sequence is of primary importance, and many reactive systems are designed not to terminate.

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. We will focus on a powerful logic called CTL* [61, 63, 105].

3.1 The Computation Tree Logic CTL*

Conceptually, CTL* formulas describe properties of *computation trees*. The tree is formed by designating a state in a Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as illustrated in Figure 3.1. The computation tree shows all of the possible executions starting from the initial state.

In CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers **A** ("for all computation paths") and **E** ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- **X** ("next time") requires that a property holds in the second state of the path.
- The **F** ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- **G** ("always" or "globally") specifies that a property holds at every state on the path.

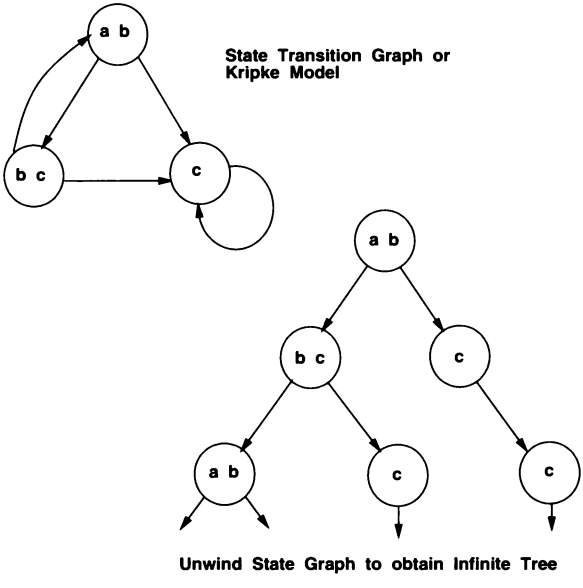


Figure 3.1
Computation trees.

- The **U** ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **R** ("release") is the logical dual of the **U** operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

The remainder of this section contains a precise description of the syntax and semantics of CTL*. There are two types of formulas in CTL*: *state formulas* (which are true in a specific state) and *path formulas* (which are true along a specific path). Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.
- If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

Two additional rules are needed to specify the syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, $f\ \mathbf{U}\ g$, and $f\ \mathbf{R}\ g$ are path formulas.

CTL* is the set of state formulas generated by the above rules.

We define the semantics of CTL* with respect to a Kripke structure. Recall that a Kripke structure M is a triple $\langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation, which must be *total* (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \to 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. A *path in M* is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. (Alternatively, we can think of a path as an infinite branch in the computation tree that corresponds to the Kripke structure.)

We use π^i to denote the *suffix* of π starting at s_i. If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in the Kripke structure M. When the Kripke structure M is clear from the context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1.	$M, s \models p$	$\Leftrightarrow$	$p \in L(s)$.
2.	$M, s \models \neg f_1$	$\Leftrightarrow$	$M, s \not\models f_1$.
3.	$M, s \models f_1 \vee f_2$	$\Leftrightarrow$	$M, s \models f_1$ or $M, s \models f_2$.
4.	$M, s \models f_1 \wedge f_2$	$\Leftrightarrow$	$M, s \models f_1$ and $M, s \models f_2$.
5.	$M, s \models \mathbf{E}\ g_1$	$\Leftrightarrow$	there is a path π from s such that $M, \pi \models g_1$.
6.	$M, s \models \mathbf{A}\ g_1$	$\Leftrightarrow$	for every path π starting from s, $M, \pi \models g_1$.
7.	$M, \pi \models f_1$	$\Leftrightarrow$	s is the first state of π and $M, s \models f_1$.
8.	$M, \pi \models \neg g_1$	$\Leftrightarrow$	$M, \pi \not\models g_1$.
9.	$M, \pi \models g_1 \vee g_2$	$\Leftrightarrow$	$M, \pi \models g_1$ or $M, \pi \models g_2$.
10.	$M, \pi \models g_1 \wedge g_2$	$\Leftrightarrow$	$M, \pi \models g_1$ and $M, \pi \models g_2$.
11.	$M, \pi \models \mathbf{X}\, g_1$	$\Leftrightarrow$	$M, \pi^1 \models g_1$.
12.	$M, \pi \models \mathbf{F}\, g_1$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $M, \pi^k \models g_1$.
13.	$M, \pi \models \mathbf{G}\, g_1$	$\Leftrightarrow$	for all $i \geq 0$, $M, \pi^i \models g_1$.
14.	$M, \pi \models g_1\ \mathbf{U}\ g_2$	$\Leftrightarrow$	there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k$, $M, \pi^j \models g_1$.
15.	$M, \pi \models g_1\ \mathbf{R}\ g_2$	$\Leftrightarrow$	for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$.

It is easy to see that the operators $\vee$, $\neg$, $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{E}$ are sufficient to express any other CTL* formula.

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $f\ \mathbf{R}\ g \equiv \neg(\neg f\ \mathbf{U}\ \neg g)$
- $\mathbf{F}\ f \equiv \mathit{True}\ \mathbf{U}\ f$
- $\mathbf{G}\ f \equiv \neg\ \mathbf{F}\ \neg f$
- $\mathbf{A}(f) \equiv \neg\ \mathbf{E}(\neg f)$

3.2 CTL and LTL

In this section we consider two useful sublogics of CTL*: one is a *branching-time* logic and one is a *linear-time* logic. The distinction between the two is in how they handle branching in the underlying computation tree. In branching-time temporal logic the temporal operators quantify over the paths that are possible from a given state. In linear-time temporal logic, operators are provided for describing events along a single computation path.

Computation Tree Logic (CTL) [19, 61, 104] is a restricted subset of CTL* in which each of the temporal operators **X**, **F**, **G**, **U**, and **R** must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL* that is obtained by restricting the syntax of path formulas using the following rule.

- If f and g are *state* formulas, then $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$, and $f\ \mathbf{R}\ g$ are path formulas.

Linear Temporal Logic (LTL) [217], on the other hand, will consist of formulas that have the form $\mathbf{A}\ f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either:

- If $p \in AP$, then p is a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$, and $f\ \mathbf{R}\ g$ are path formulas.

It can be shown [59, 105, 166] that the three logics that we have discussed have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}\ p)$. This formula expresses the property that along every path, there is some state from which p will hold forever. Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}\ p)$. The disjunction of these two formulas $\mathbf{A}(\mathbf{FG}\ p) \vee \mathbf{AG}(\mathbf{EF}\ p)$ is a CTL* formula that is not expressible in either CTL or LTL.

Most of the specifications in this book will be written in the logic CTL. There are ten basic CTL operators:

- **AX** and **EX**,

- **AF** and **EF**,
- **AG** and **EG**
- **AU** and **EU**,
- **AR** and **ER**.

Each of the ten operators can be expressed in terms of three operators **EX**, **EG**, and **EU**:

- $\mathbf{AX}\ f = \neg\ \mathbf{EX}(\neg f)$
- $\mathbf{EF}\ f = \mathbf{E}[\mathit{True}\ \mathbf{U}\ f]$
- $\mathbf{AG}\ f = \neg\ \mathbf{EF}(\neg f)$
- $\mathbf{AF}\ f = \neg\ \mathbf{EG}(\neg f)$
- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\ \mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\ \mathbf{EG}\ \neg g$
- $\mathbf{A}[f\ \mathbf{R}\ g] \equiv \neg\ \mathbf{E}[\neg f\ \mathbf{U}\ \neg g]$
- $\mathbf{E}[f\ \mathbf{R}\ g] \equiv \neg\ \mathbf{A}[\neg f\ \mathbf{U}\ \neg g]$

The four operators that are used most widely are illustrated in Figure 3.2. The operators are easiest to understand in terms of the computation tree obtained by unfolding the Kripke model. Each computation tree has the state s_0 as its root.

Some typical CTL formulas that might arise in verifying a finite state concurrent program are given below:

- $\mathbf{EF}(\mathit{Start} \wedge \neg \mathit{Ready})$: It is possible to get to a state where *Start* holds but *Ready* does not hold.
- $\mathbf{AG}(\mathit{Req} \rightarrow \mathbf{AF}\ \mathit{Ack})$: If a request occurs, then it will be eventually acknowledged.
- $\mathbf{AG}(\mathbf{AF}\ \mathit{DeviceEnabled})$: The proposition *DeviceEnabled* holds infinitely often on every computation path.
- $\mathbf{AG}(\mathbf{EF}\ \mathit{Restart})$: From any state it is possible to get to the *Restart* state.

Many of the methods to avoid the state explosion problem rely on compositional reasoning or abstraction. The logic that is typically used in these cases is more restricted and allows only *universal path quantifiers*. The restriction of CTL* to universal path quantifiers is called ACTL*, and the restriction of CTL to universal path quantifiers is called ACTL.

In order to avoid implicit existential path quantifiers resulting from the use of negation, we assume that the formulas are given in *positive normal form*, that is, negations are applied only to atomic propositions. To avoid the loss of expressive power, we need conjunction and disjunction, and both the **U** and **R** operators.

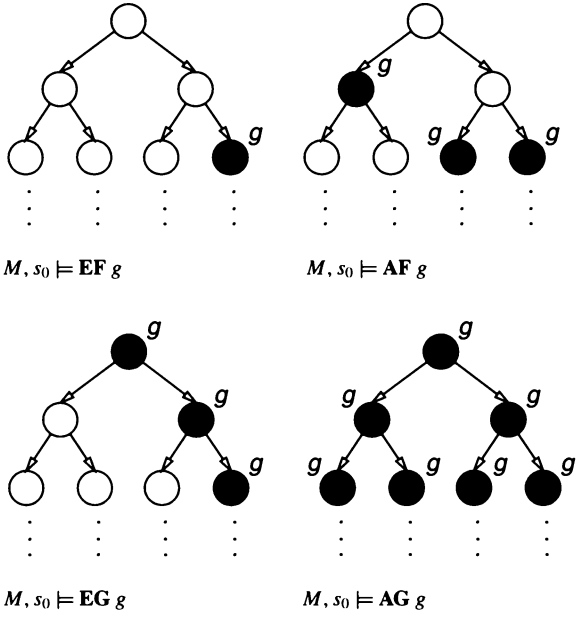


Figure 3.2
Basic CTL operators.

The formulas **AF AG** a and **AF AX** a are examples of ACTL formulas. These formulas are not expressible in LTL [59]. Because ACTL is a subset of CTL, the logics ACTL and LTL are incomparable. Moreover, ACTL* is more expressive than LTL. The formulas **AG EF** *Start* and **AG** $\neg$ **AF** *Start* are not in ACTL.

3.3 Fairness

Finally, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs forever. Alternatively, we may want to consider communication protocols that operate over reliable channels which have the property that no message is ever continuously transmitted but never received. Such properties cannot be expressed directly in CTL [59, 104, 105] but can be expressed in CTL*. In order to deal with fairness in CTL we must modify its semantics slightly. We call the new semantics of the logic the *fair semantics*. A *fairness constraint* can be an arbitrary set of states, usually

described by a formula of the logic. If fairness constraints are interpreted as sets of states, then a fair path must contain an element of each fairness constraint infinitely often. If fairness constraints are interpreted as CTL formulas, then a path is *fair* if each constraint is true *infinitely often* along the path. The path quantifiers in the logic are then restricted to fair paths.

Formally, a *fair Kripke structure* is a 4-tuple $M = (S, R, L, F)$, where S, L, and R are defined as before and $F \subseteq 2^S$ is a set of fairness constraints (often called generalized Büchi acceptance conditions). Let $\pi = s_0, s_1, \ldots$ be a path in M. Define

$$\inf(\pi) = \{ s \mid s = s_i \text{ for infinitely many } i \}.$$

We say that π is *fair* if and only if for every $P \in F$, $\inf(\pi) \cap P \neq \emptyset$. The semantics of CTL* with respect to a fair Kripke structure is very similar to the semantics of CTL* with respect to an ordinary Kripke structure. We will write $M, s \models_F f$ to indicate that the state formula f is true in state s of the fair Kripke structure M. Similarly, we write $M, \pi \models_F g$ to indicate that the path formula g is true along path π in M. Only clauses 1, 5 and 6 in the original semantics change.

1.	$M, s \models_F p$	$\Leftrightarrow$	there exists a fair path starting from s and $p \in L(s)$.
5.	$M, s \models_F \mathbf{E}(g_1)$	$\Leftrightarrow$	there exists a fair path π starting from s such that $\pi \models_F g_1$.
6.	$M, s \models_F \mathbf{A}(g_1)$	$\Leftrightarrow$	for all fair paths π starting from s, $\pi \models_F g_1$.

To illustrate the use of fairness, consider again the communication protocol for reliable channels. There is one fairness constraint for each channel that expresses the reliability of that channel. A possible choice for the fairness constraint associated with channel i is the set of states that satisfy the formula $\neg send_i \vee receive_i$. Thus, a computation path is fair if and only if for every channel, infinitely often either a message is not sent or a message is received. Other notions of fairness are dealt with in [116].

4 Model Checking

The *model checking problem* is easy to describe. Given a Kripke structure $M = (S, R, L)$ that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, find the set of all states in S that satisfy f:

$$\{ s \in S \mid M, s \models f \}.$$

Normally, some states of the concurrent system are designated as *initial states*. The system satisfies the specification provided that all of the initial states are in the set.

The first algorithms for solving the model checking problem used an *explicit* representation of the Kripke structure as a labeled, directed graph with arcs given by pointers. In this case, the nodes represent the states in S, the arcs in the graph give the transition relation R, and the labels associated with the nodes describe the function $L : S \to 2^{AP}$.

4.1 CTL Model Checking

Let $M = (S, R, L)$ be a Kripke structure. Assume that we want to determine which states in S satisfy the CTL formula f. The algorithm will operate by labeling each state s with the set $label(s)$ of subformulas of f which are true in s. Initially, $label(s)$ is just $L(s)$. The algorithm then goes through a series of stages. During the ith stage, subformulas with $i - 1$ nested CTL operators are processed. When a subformula is processed, it is added to the labeling of each state in which it is true. Once the algorithm terminates, we will have that $M, s \models f$ iff $f \in label(s)$.

Recall that any CTL formula can be expressed in terms of $\neg$, $\vee$, **EX**, **EU** and **EG**. Thus, for the intermediate stages of the algorithm it is sufficient to be able to handle six cases, depending on whether g is atomic or has one of the following forms: $\neg f_1$, $f_1 \vee f_2$, $\mathbf{EX}\ f_1$, $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$, or $\mathbf{EG}\ f_1$.

For formulas of the form $\neg f_1$, we label those states that are not labeled by f_1. For $f_1 \vee f_2$, we label any state that is labeled either by f_1 or by f_2. For $\mathbf{EX}\ f_1$, we label every state that has some successor labeled by f_1.

To handle formulas of the form $g = \mathbf{E}[f_1\ \mathbf{U}\ f_2]$ we first find all states that are labeled with f_2. We then work backwards using the converse of the transition relation R and find all states that can be reached by a path in which each state is labeled with f_1. All such states should be labeled with g.

In Figure 4.1 we give a procedure $CheckEU$ that adds $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ to $label(s)$ for every s that satisfies $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$, assuming that f_1 and f_2 have already been processed correctly, that is, for every state s, $f_1 \in label(s)$ iff $s \models f_1$ and $f_2 \in label(s)$ iff $s \models f_2$. This procedure requires time $O(|S| + |R|)$.

```
procedure CheckEU(f1, f2)
    T := { s | f2 ∈ label(s) };
    for all s ∈ T do label(s) := label(s) ∪ { E[f1 U f2] };
    while T ≠ ∅ do
        choose s ∈ T;
        T := T \ {s};
        for all t such that R(t, s) do
            if E[f1 U f2] ∉ label(t) and f1 ∈ label(t) then
                label(t) := label(t) ∪ { E[f1 U f2] };
                T := T ∪ {t};
            end if;
        end for all;
    end while;
end procedure
```

Figure 4.1
Procedure for labeling the states satisfying $\mathbf{E}[f_1 \mathbf{U} f_2]$.

The case in which $g = \mathbf{EG}\ f_1$ is slightly more complicated. It is based on the decomposition of the graph into nontrivial strongly connected components. A *strongly connected component* (SCC) C is a *maximal* subgraph such that every node in C is reachable from every other node in C along a directed path entirely contained within C. C is *nontrivial* iff either it has more than one node or it contains one node with a self-loop.

Let M' be obtained from M by deleting from S all of those states at which f_1 does not hold and restricting R and L accordingly. Thus, $M' = (S', R', L')$ where $S' = \{ s \in S \mid M, s \models f_1 \}$, $R' = R|_{S' \times S'}$, and $L' = L\ |_{S'}$. Note that R' may not be total in this case. The states with no outgoing transitions may be eliminated, but this is not essential for the correctness of our algorithm. The algorithm depends on the following observation.

LEMMA 1 $M, s \models \mathbf{EG}\ f_1$ iff the following two conditions are satisfied:

1. $s \in S'$.
2. There exists a path in M' that leads from s to some node t in a nontrivial strongly connected component C of the graph (S', R').

Proof Assume that $M, s \models \mathbf{EG}\ f_1$. Clearly $s \in S'$. Let π be an infinite path starting at s such that f_1 holds at each state on π. Since M is finite, it must be possible to write π as $\pi = \pi_0 \pi_1$ where π_0 is a finite initial segment and π_1 is an infinite suffix of π with the property that each state on π_1 occurs infinitely often. Then, π_0 is contained in S'. Let C be the set of states in π_1. Clearly, C is contained in S'. We now show that there

```
procedure CheckEG(f1)
    S' := { s | f1 ∈ label(s) };
    SCC := { C | C is a nontrivial SCC of S' };
    T := ⋃_{C∈SCC} { s | s ∈ C };
    for all s ∈ T do label(s) := label(s) ∪ { EG f1 };
    while T ≠ ∅ do
        choose s ∈ T;
        T := T \ {s};
        for all t such that t ∈ S' and R(t, s) do
            if EG f1 ∉ label(t) then
                label(t) := label(t) ∪ { EG f1 };
                T := T ∪ {t};
            end if;
        end for all;
    end while;
end procedure
```

Figure 4.2
Procedure for labeling the states satisfying **EG** f_1.

is a path within C between any pair of states in C. Let s_1 and s_2 be states in C. Pick some instance of s_1 on π_1. By the way in which π_1 was selected, we know that there is an instance of s_2 further along π_1. The segment from s_1 to s_2 lies entirely within C. This segment is a finite path from s_1 to s_2 in C. Thus, either C is a strongly connected component or it is contained within one. In either case, both conditions (1) and (2) are satisfied.

Next, assume that Conditions (1) and (2) are satisfied. Let π_0 be the path from s to t. Let π_1 be a finite path of length at least 1 that leads from t back to t. The existence of π_1 is guaranteed because t is a state in a nontrivial strongly connected component. All the states on the infinite path $\pi = \pi_0\pi_1^\omega$ satisfy f_1. Since π is also a possible path starting at s in M, we see that $M, s \models \mathbf{EG}\ f_1$. □

The algorithm for the case of $g = \mathbf{EG}\ f_1$ follows directly from the lemma. We construct the restricted Kripke structure $M' = (S', R', L')$ as described above. We partition the graph (S', R') into strongly connected components using the algorithm of Tarjan [2]. This algorithm has time complexity $O(|S'| + |R'|)$. Next, we find those states that belong to nontrivial components. We then work backward using the converse of R' and find all of those states that can be reached by a path in which each state is labeled with f_1. The entire computation can be performed in time $O(|S| + |R|)$. In Figure 4.2 we give a procedure

CheckEG that adds **EG** f_1 to *label*(s) for every s that satisfies **EG** f_1, assuming that f_1 has already been processed correctly.

In order to handle an arbitrary CTL formula f, we successively apply the state-labeling algorithm to the subformulas of f, starting with the shortest, most deeply nested, and work outward to include all of f. By proceeding in this manner we guarantee that whenever we process a subformula of f all its subformulas have already been processed. Since each pass takes time $O(|S| + |R|)$ and since f has at most $|f|$ different subformulas, the entire algorithm requires time $O(|f| \cdot (|S| + |R|))$.

THEOREM 1 There is an algorithm for determining whether a CTL formula f is true in a state s of the structure $M = (S, R, L)$ that runs in time $O(|f| \cdot (|S| + |R|))$.

We will illustrate the model checking algorithm for CTL on a small example that describes the behavior of a microwave oven. Figure 4.3 gives the Kripke structure for the oven. For clarity, each state is labeled with both the atomic propositions that are true in the state and the negations of the propositions that are false in the state. The labels on the arcs indicate the actions that cause transitions and are not part of the Kripke structure.

We check the CTL formula $\mathbf{AG}(Start \rightarrow \mathbf{AF}\ Heat)$ which is equivalent to the formula $\neg\ \mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat)$ (here, we use **EF** f as an abbreviation for $\mathbf{E}[true\ \mathbf{U}\ f]$). We start by computing the set of states that satisfy the atomic formulas and proceed to more complicated subformulas. Let $S(g)$ denote the set of all states labeled by the subformula g. Note that, with a suitable data structure, the computation of $S(p)$ for all $p \in AP$ requires time $O(|S| + |R|)$.

$S(Start) = \{2, 5, 6, 7\}$.

$S(\neg Heat) = \{1, 2, 3, 5, 6\}$.

In order to compute $S(\mathbf{EG}\ \neg Heat)$ we first find the set of nontrivial strongly connected components in $S' = S(\neg Heat)$. $SCC = \{\{1, 2, 3, 5\}\}$. We proceed by setting T, the set of all states that should be labeled by $\mathbf{EG}\ \neg Heat$ to be the union over the elements of SCC, that is, initially $T = \{1, 2, 3, 5\}$. No other state in S' can reach a state in T along a path in S'. Thus, the computation terminates with

$S(\mathbf{EG}\ \neg Heat) = \{1, 2, 3, 5\}$.

Next we compute

$S(Start \wedge \mathbf{EG}\ \neg Heat) = \{2, 5\}$.

When computing $S(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat))$, we start by setting $T = S(Start \wedge$

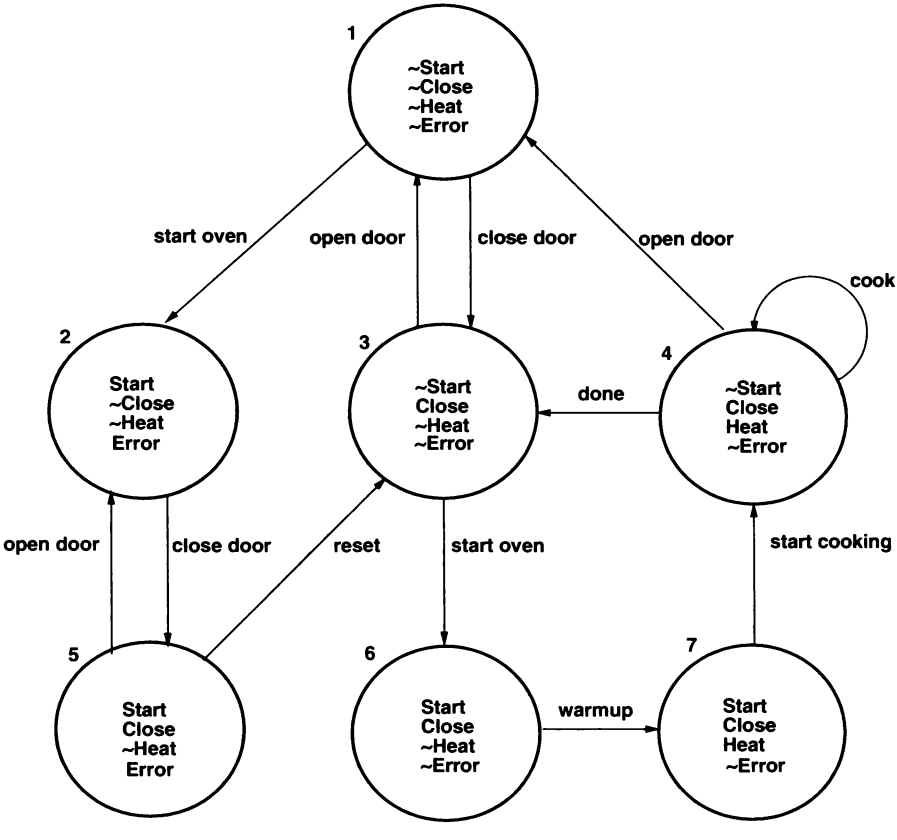


Figure 4.3
Microwave oven example.

$\mathbf{EG}\ \neg Heat$). Next, we use the converse of the transition relation to label all states in which the formula holds. We get:

$S(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat)) = \{1, 2, 3, 4, 5, 6, 7\}$.

Finally, we compute that

$S(\neg\ \mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat)) = \emptyset$.

Since the initial state 1 is not contained in this set, we conclude that the system described by the Kripke structure does not satisfy the given specification.

4.1.1 Fairness Constraints

In this subsection, we show how to extend the CTL model checking algorithm to handle fairness constraints. Let $M = (S, R, L, F)$ be a fair Kripke structure. Let $F = \{P_1, \ldots, P_k\}$ be the set of fairness constraints. We will say that a strongly connected component C of the graph of M is *fair* with respect to F if and only if for each $P_i \in F$, there is a state $t_i \in (C \cap P_i)$. We first give an algorithm for checking $\mathbf{EG}\ f_1$ with respect to a fair structure. In order to establish the correctness of this algorithm, we need a lemma that is analogous to Lemma 1. As before, let M' be obtained from M by deleting from S all of those states at which f_1 does not *fairly* hold. Thus, $M' = (S', R', L', F')$, where $S' = \{\, s \in S \mid M, s \models_F f_1 \,\}$, $R' = R|_{S' \times S'}$, $L' = L|_{S'}$, and $F' = \{\, P_i \cap S' \mid P_i \in F \,\}$.

LEMMA 2 $M, s \models_F \mathbf{EG}\ f_1$ iff the following two conditions are satisfied:

1. $s \in S'$.
2. There exists a path in S' that leads from s to some node t in a nontrivial *fair* strongly connected component of the graph (S', R').

The proof of this lemma is similar to the proof of Lemma 1 and will not be given. We can now describe the procedure $CheckFairEG(f_1)$ that adds $\mathbf{EG}\ f_1$ to the label of s for every s such that $M, s \models_F \mathbf{EG}\ f_1$. We assume that the states have been labeled correctly with f_1 using the fair semantics for the logic, that is, we assume $f_1 \in label(s)$ if and only if $M, s \models_F f_1$. The procedure $CheckFairEG$ is identical to the procedure $CheckEG$ given in Figure 4.2. The only difference is that SCC now consists of the set of nontrivial *fair* strongly connected components. The complexity of this computation is $O((|S| + |R|) \cdot |F|)$ since it is necessary to determine which strongly connected components are fair. This involves examining every component to see if it has a state from each fairness constraint.

In order to check other CTL formulas with respect to fair Kripke structures we introduce an additional atomic proposition $fair$, which is true at a state if and only if there is a fair path starting from that state. Thus, we have that $fair = \mathbf{EG}\ true$ according to the fair semantics for the logic. The procedure $CheckFairEG(true)$ can be used to label states with the new atomic proposition. In order to determine if $M, s \models_F p$ for some $p \in AP$, we check $M, s \models p \wedge fair$ using the ordinary model-checking procedure. In order to determine if $M, s \models_F \mathbf{EX}\ f_1$ we check $M, s \models \mathbf{EX}(f_1 \wedge fair)$. Finally, in order to determine if $M, s \models_F \mathbf{E}[f_1\ \mathbf{U}\ f_2]$ we check $M, s \models \mathbf{E}[f_1\ \mathbf{U}\ (f_2 \wedge fair)]$ by calling the procedure $CheckEU(f_1, f_2 \wedge fair)$.

The complexity analysis is similar to the nonfair case. Each stage requires time $O((|S| + |R|) \cdot |F|)$. Since there are at most $|f|$ stages, the total time complexity is $O(|f| \cdot (|S| + |R|) \cdot |F|)$.

THEOREM 2 There is an algorithm for determining whether a CTL formula f is true with respect to the fair semantics in a state s of the structure $M = (S, R, L, F)$ that runs in time $O(|f| \cdot (|S| + |R|) \cdot |F|)$.

To illustrate the use of fairness constraints, we again check the formula $\mathbf{AG}(Start \rightarrow \mathbf{AF}\ Heat)$ on the model described in Figure 4.3. However, this time we consider only paths along which the user operates the microwave oven correctly infinitely often. This means that infinitely often $Start \wedge Close \wedge \neg Error$ should hold. Thus, $F = \{P\}$, where $P = \{s | s \models Start \wedge Close \wedge \neg Error\}$. $S(Start)$ and $S(\neg Heat)$ remain as before. When we compute the set of strongly connected components over $S' = S(\neg Heat)$, we realize that $\{1, 2, 3, 5\}$ is not fair since it does not contain a state that satisfies $Start \wedge Close \wedge \neg Error$. Thus,

$$S(\mathbf{EG}\ \neg Heat) = \emptyset.$$

As a result we get

$$S(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat)) = \emptyset$$

which implies that

$$S(\neg(\mathbf{EF}(Start \wedge \mathbf{EG}\ \neg Heat))) = \{1, 2, 3, 4, 5, 6, 7\}.$$

Thus, the program satisfies the formula under the given fairness constraints.

4.2 LTL Model Checking by Tableau

Let $M = (S, R, L)$ be a Kripke structure with $s \in S$ and let $\mathbf{A}\ g$ be a linear temporal logic formula. Thus, g is a *restricted path formula* in which the only state subformulas are atomic propositions. We wish to determine if $M, s \models \mathbf{A}\ g$. Notice that $M, s \models \mathbf{A}\ g$ if and only if $M, s \models \neg\ \mathbf{E}\ \neg g$. Consequently, it is sufficient to be able to check the truth of formulas of the form $\mathbf{E}\ f$ where f is a restricted path formula. In general, this problem is **PSPACE**-complete [232, 233]. The proof of this **PSPACE**-completeness result is quite technical and will not be given in this book. However, we will show that the model-checking problem is **NP**-hard for formulas of the form $\mathbf{E}\ f$ where f is a restricted path formula. Consider an arbitrary directed graph $G = (V, A)$ where $V = \{v_1, \ldots, v_n\}$. We show that the problem of determining whether G has a directed Hamiltonian path is reducible to the problem of determining whether $M, s \models f$ where

- M is a finite Kripke structure,
- s is a state in M and

- f is the formula (using atomic propositions $p_1, \ldots, p_n$):

$$\mathbf{E}[\mathbf{F}\, p_1 \wedge \cdots \wedge \mathbf{F}\, p_n \wedge \mathbf{G}(p_1 \rightarrow \mathbf{X}\,\mathbf{G}\,\neg p_1) \wedge \cdots \wedge \mathbf{G}(p_n \rightarrow \mathbf{X}\,\mathbf{G}\,\neg p_n)].$$

We obtain a structure from G by making propositions p_i true at node v_i and false at all other nodes (for $1 \leq i \leq n$), and by adding a source node u_1 from which all v_i are accessible (but not vice versa) and a sink node u_2 that is accessible from all v_i (but not vice versa). The source node provides a unique initial state that lets us begin the Hamiltonian path at any node of the graph. The sink node is needed to ensure that the transition relation of the structure is total. Formally, let the Kripke structure $M = (U, B, L)$ consist of

- $U = V \cup \{u_1, u_2\}$ where $u_1, u_2 \notin V$.
- $B = A \cup \{\, (u_1, v_i) \mid v_i \in V \,\} \cup \{\, (v_i, u_2) \mid v_i \in V \,\} \cup \{(u_2, u_2)\}$; and
- L is an assignment of propositions to states such that
- p_i is true in v_i for $1 \leq i \leq n$
- p_j is false in v_i for $1 \leq i, j \leq n, i \neq j$
- p_i is false in u_1, u_2 for $1 \leq i \leq n$.

It is easy to see that $M, u_1 \models f$ if and only if there is a directed infinite path in M starting at u_1 that goes through all $v_i \in V$ exactly once and ends in the self loop through u_2.

Note that the formula f in the above construction has essentially the same size as the graph G. Suppose that the length of the formula to be checked is much smaller than the size of the Kripke structure under consideration. Would the complexity still be high in this case? A careful analysis by Lichtenstein and Pnueli [173] shows that although the complexity is apparently exponential in the length of the formula, it is linear in the size of the global state graph. Their algorithm involves an implicit *tableau construction*. A tableau is a graph derived from the formula from which a model for the formula can be extracted if and only if the formula is satisfiable. The algorithm to check whether M satisfies f composes the tableau and the structure in order to determine whether there exists a computation of the structure that is a path in the tableau. Below, we describe the algorithm of [173]. In section 6.7, we present a different model checking algorithm for LTL that involves a direct tableau construction.

Let $M = (S, R, L)$ be a Kripke structure and let f be a restricted path formula. It is sufficient to consider only the temporal operators $\mathbf{X}$ and $\mathbf{U}$, since $\mathbf{F}\, f = \mathit{True}\, \mathbf{U}\, f$, $\mathbf{G}\, f = \neg\, \mathbf{F}\, \neg f$ and $f_1\, \mathbf{R}\, f_2 = \neg[\neg f_1\, \mathbf{U}\, \neg f_2]$. The *closure* of f, $CL(f)$, contains formulas whose truth values can influence the truth value of f. More precisely, it is the smallest set of formulas containing f and satisfying:

- $\neg f_1 \in CL(f)$ iff $f_1 \in CL(f)$

- if $f_1 \vee f_2 \in CL(f)$, then $f_1, f_2 \in CL(f)$
- if $\mathbf{X}\, f_1 \in CL(f)$, then $f_1 \in CL(f)$
- if $\neg\, \mathbf{X}\, f_1 \in CL(f)$, then $\mathbf{X} \neg f_1 \in CL(f)$
- if $f_1\, \mathbf{U}\, f_2 \in CL(f)$, then $f_1, f_2, \mathbf{X}[f_1\, \mathbf{U}\, f_2] \in CL(f)$.

(In the above, we identify $\neg\neg f_1$ with f_1.) It can be shown that the size of $CL(f)$ is linear in the size of f. An *atom* is a pair $A = (s_A, K_A)$ with $s_A \in S$ and $K_A \subseteq CL(f) \cup AP$ such that:

- for each proposition $p \in AP$, $p \in K_A$ iff $p \in L(s_A)$
- for every $f_1 \in CL(f)$, $f_1 \in K_A$ iff $\neg f_1 \notin K_A$
- for every $f_1 \vee f_2 \in CL(f)$, $f_1 \vee f_2 \in K_A$ iff $f_1 \in K_A$ or $f_2 \in K_A$
- for every $\neg\, \mathbf{X}\, f_1 \in CL(f)$, $\neg\, \mathbf{X}\, f_1 \in K_A$ iff $\mathbf{X} \neg f_1 \in K_A$
- for every $f_1\, \mathbf{U}\, f_2 \in CL(f)$, $f_1\, \mathbf{U}\, f_2 \in K_A$ iff $f_2 \in K_A$ or $f_1, \mathbf{X}[f_1\, \mathbf{U}\, f_2] \in K_A$.

Intuitively, an atom (s_A, K_A) is defined so that K_A is a maximal consistent set of formulas that is also consistent with the labeling of s_A.

A graph G is constructed with the set of atoms as the set of vertices. (A, B) is an edge of G iff $(s_A, s_B) \in R$ and for every formula $\mathbf{X}\, f_1 \in CL(f)$, $\mathbf{X}\, f_1 \in K_A$ iff $f_1 \in K_B$. An *eventuality sequence* is an infinite path π in G such that if $f_1\, \mathbf{U}\, f_2 \in K_A$ for some atom A on π, then there exists an atom B, reachable from A along π, such that $f_2 \in K_B$.

LEMMA 3 $M, s \models \mathbf{E}\, f$ iff there exists an eventuality sequence starting at an atom (s, K) such that $f \in K$.

Proof We only sketch the proof here. First assume that there is an eventuality sequence $(s_0, K_0), (s_1, K_1), \ldots$ starting at $(s, K) = (s_0, K_0)$ with $f \in K$. By definition, $\pi = s_0, s_1, \ldots$ is a path in M starting at $s = s_0$. We want to show that $\pi \models f$. Actually, we prove a stronger claim: for every $g \in CL(f)$ and every $i \geq 0$, $\pi^i \models g$ iff $g \in K_i$. The proof proceeds by induction on the structure of the subformulas. Here we give the base case and the inductive step when g is either $\neg h_1$, $h_1 \vee h_2$, $\mathbf{X}\, h_1$, or $h_1\, \mathbf{U}\, h_2$.

1. If g is an atomic proposition, then by the definition of an atom, $g \in K_i$ iff $g \in L(s_i)$.
2. If $g = \neg h_1$ then $\pi^i \models g$ iff $\pi^i \not\models h_1$. By the induction hypothesis, this is true iff $h_1 \notin K_i$. By the definition of K_i, this guarantees that $g \in K_i$.
3. If $g = h_1 \vee h_2$ then $\pi^i \models g$ iff $\pi^i \models h_1$ or $\pi^i \models h_2$. By the induction hypothesis this holds iff $h_1 \in K_i$ or $h_2 \in K_i$. By the definition of K_i, this is true iff $g \in K_i$.

4. If $g = \mathbf{X}\, h_1$ then $\pi^i \models g$ iff $\pi^{i+1} \models h_1$. By the induction hypothesis this holds iff $h_1 \in K_{i+1}$. Since $((s_i, K_i), (s_{i+1}, K_{i+1})) \in R$, the above holds iff $\mathbf{X}\, h_1 \in K_i$.

5. Suppose $g = h_1 \,\mathbf{U}\, h_2 \in K_i$. By the definition of an eventually sequence, there is some $j \geq i$ such that $h_2 \in K_j$. Since $g \in K_i$, the definition of an atom implies that if $h_2 \notin K_i$, then $h_1 \in K_i$ and $\mathbf{X}\, g \in K_i$. In this case, the definition of the transition relation of G then implies that $g \in K_{i+1}$. It follows that for every $i \leq k < j$, $h_1 \in K_k$. By the induction hypothesis, $\pi^j \models h_2$ and for every $i \leq k < j$, $\pi^k \models h_1$. Hence $\pi^i \models g$.

If $\pi^i \models g$, then there exists $j \geq i$ such that $\pi^j \models h_2$ and for all $i \leq k < j$, $\pi^k \models h_1$. Choose the minimum such j. By the induction hypothesis, $h_2 \in K_j$ and for every $i \leq k < j$, $h_1 \in K_k$. Suppose $g \notin K_i$. Because $h_1 \in K_i$, by the definition of an atom $\mathbf{X}\, g \notin K_i$, which implies that $\mathbf{X}\, \neg g \in K_i$. Now by the definition of the transition relation of G, $\neg g \in K_{i+1}$, and hence $g \notin K_{i+1}$. Continuing the argument inductively, we would eventually find $g \notin K_j$, which is a contradiction since $h_2 \in K_j$.

This concludes the first part of the proof.

For the other direction, assume that $M, s \models \mathbf{E}\, f$. Then there is a path $\pi = s_0, s_1, \ldots$ from $s = s_0$ such that $\pi \models f$. Define $K_i = \{\, g \mid g \in CL(f) \text{ and } \pi^i \models g \,\}$. Thus, the following hold:

1. (s_i, K_i) is an atom. This follows from the definition of $\models$. For example, given $g \in CL(f)$, an atom K_i should contain either g or $\neg g$, but not both. By the definition of K_i, $g \in K_i$ iff $\pi^i \models g$. But $\pi^i \models g$ iff $\pi^i \not\models \neg g$. Again by definition, $\pi^i \not\models \neg g$ iff $\neg g \notin K_i$.

2. There is a transition from (s_i, K_i) to (s_{i+1}, K_{i+1}). This follows from the observation that $\mathbf{X}\, g \in K_i$ iff $\pi^i \models \mathbf{X}\, g$. Further, $\pi^i \models \mathbf{X}\, g$ iff $\pi^{i+1} \models g$. Finally, by the definition of K_{i+1}, $\pi^{i+1} \models g$ iff $g \in K_{i+1}$. Thus, $\mathbf{X}\, g \in K_i$ iff $g \in K_{i+1}$.

3. The sequence $(s_0, K_0), (s_1, K_1), \ldots$ is an eventuality sequence. Note that $g = h_1 \,\mathbf{U}\, h_2 \in K_i$ iff $\pi^i \models g$. This means that there is some $j \geq i$ such that $\pi^j \models h_2$, which tells us that $h_2 \in K_j$. □

A nontrivial strongly connected component C of the graph G is said to be *self-fulfilling* iff for every atom A in C and for every $f_1 \,\mathbf{U}\, f_2 \in K_A$ there exists an atom B in C such that $f_2 \in K_B$.

LEMMA 4 There exists an eventuality sequence starting at an atom (s, K) iff there is a path in G from (s, K) to a self-fulfilling strongly connected component.

Proof Assume that there is an eventuality sequence starting at (s, K). Consider the set C' of all atoms that appear infinitely often in this sequence. The set C' is a subset of a (maximal) strongly connected component C of G. Consider a subformula $g = h_1 \,\mathbf{U}\, h_2$ and

an atom $(s, K) \in C$ such that $g \in K$. Because C is strongly connected, there is a finite path in C from (s, K) into C'. If h_2 appears on this path, there is clearly an atom in C containing h_2. If h_2 does not appear on the path, then g is in every atom on the path, and in particular, g is in an atom of C'. Since C' comes from an eventuality sequence, h_2 is in some atom of C', and hence of C. Thus C is self-fulfilling.

Now suppose that there is a path from (s, K) to a self-fulfilling strongly connected component C. Clearly we can construct a sequence within C where every subformula $h_1 \mathbf{U} h_2$ is followed by an occurrence of h_2. The only question concerns subformulas $h_1 \mathbf{U} h_2$ that appear on the path from (s, K) to C. Each such subformula must either be followed by an occurrence of h_2, or it remains in all atoms along the path until C is reached. Then since C is self-fulfilling, we will be able to reach an occurrence of h_2. □

COROLLARY 1 $M, s \models \mathbf{E}\, f$ iff there exists an atom $A = (s, K)$ in G such that $f \in K$ and there exists a path in G from A to a self-fulfilling strongly connected component.

Corollary 1 can be used as the basis for a linear temporal logic model checking algorithm. This algorithm has the time complexity $O((|S| + |R|) \cdot 2^{O(|f|)})$. Lichtenstein and Pnueli further show how this basic algorithm can be extended to handle a number of different notions of fairness with essentially the same complexity.

To demonstrate how the LTL model checking works, consider again the model $M = (S, R, L)$ of Figure 4.3 with the specification $\mathbf{A}[(\neg Heat)\ \mathbf{U}\ Close]$ which will be true in the model if it is impossible for the oven to be hot with the door open. In order to show that this formula is satisfied we will check that its negation, $\mathbf{E}\,\neg((\neg Heat)\ \mathbf{U}\ Close)$, is not satisfied. Let f denote $(\neg Heat)\ \mathbf{U}\ Close$. We first compute the closure of $\neg f$

$$CL(\neg f) = \{\neg f,\ f,\ \mathbf{X}\, f,\ \neg \mathbf{X}\, f,\ \mathbf{X} \neg f,\ Heat,\ \neg Heat,\ Close,\ \neg Close\}.$$

Next, we construct the set of atoms that will constitute the vertices of the graph G. According to the last clause of the definition of K_A, $(\neg Heat)\ \mathbf{U}\ Close$ is in K_A iff either $Close$ is in K_A or both $\neg Heat$ and $\mathbf{X}((\neg Heat)\ \mathbf{U}\ Close)$ are in K_A. Also, K_A must be consistent with $L(s_A)$. The formulas $\neg Close$ and $\neg Heat$ are contained in the labels of states 1 and 2. Therefore, the set of formulas associated with these states can have one of the following forms:

$$K_1' = \{\neg Close, \neg Heat, f, \mathbf{X}\, f\} \quad \text{or} \quad K_1'' = \{\neg Close, \neg Heat, \neg f, \mathbf{X} \neg f, \neg \mathbf{X}\, f\}.$$

Thus, $(1, K_1')$, $(2, K_1')$, $(1, K_1'')$, and $(2, K_1'')$ are atoms. Similarly, the labels of states 3, 5, and 6 contain $\neg Heat$ and $Close$. Therefore, the two sets that can be associated with them are:

$$K_2' = \{Close, \neg Heat, f, \mathbf{X}\, f\} \quad \text{or} \quad K_2'' = \{Close, \neg Heat, f, \mathbf{X} \neg f, \neg \mathbf{X}\, f\}.$$

Finally, for states 4 and 7 these sets are:

$$K_3' = \{Close, Heat, f, \mathbf{X}\, f\} \quad \text{or} \quad K_3'' = \{Close, Heat, f, \mathbf{X} \neg f, \neg \mathbf{X}\, f\}.$$

To define the transition relation between atoms we recall that there is a transition from atom (s_A, K_A) to atom (s_B, K_B) if there is a transition from s_A to s_B in M and moreover, for every formula of the form $\mathbf{X}\, f \in K_A$, $f \in K_B$. Since $(1, 2) \in R$, there is a transition from $(1, K_1')$ to $(2, K_1')$ since $\mathbf{X}\, f \in K_1'$ and $f \in K_1'$. There is also a transition from $(1, K_1'')$ to $(2, K_1'')$ since $\mathbf{X} \neg f \in K_1''$ and $\neg f \in K_1''$. However, there is no transition from $(1, K_1')$ to $(2, K_1'')$ since $\mathbf{X}\, f \in K_1'$ but $f \notin K_1''$.

The rest of the transition relation is constructed accordingly. By Corollary 1, a state s satisfies $\neg f$ if there is an atom (s, K) such that $\neg f \in K$ and there is a path from (s, K) in G that leads to a self-fulfilling strongly connected component. Once the full graph is constructed, it is easy to see that no such atom is the beginning of an infinite path. Thus, no state satisfies $\mathbf{E} \neg f$. This implies that all states satisfy $\mathbf{A}\, f$.

4.3 CTL* Model Checking

One would expect that the complexity of the model checking problem for CTL* should be greater than the complexity of the model checking problems for both CTL and LTL. Surprisingly, this is not the case. In [62, 106] it is shown that the model checking problem for CTL* has essentially the same complexity as the model checking problem for LTL.

The basic idea is to combine the state labeling technique from CTL model checking with LTL model checking. The original algorithm for LTL can handle formulas of the form $\mathbf{E}\, f$ where f is a restricted path formula in which the only state subformulas are atomic propositions. This algorithm can be extended to handle formulas in which f contains arbitrary state subformulas. Assume that the state subformulas of f have already been processed and that the state labels have been updated accordingly. Each state subformula will be replaced by a fresh atomic proposition in both the labeling of the model and the formula. Let the new formula be denoted by $\mathbf{E}\, f'$. If the formula is in CTL, then we apply the CTL model checking procedure. Otherwise, f' is a pure LTL path formula, and the algorithm for LTL model checking is used. In both cases, the formula is added to the label of all of those states that satisfy it. If $\mathbf{E}\, f$ is a subformula of a more complex CTL* formula, then the procedure is repeated with $\mathbf{E}\, f$ replaced by a fresh atomic proposition. This is continued until the entire formula is processed.

Like the CTL algorithm, the algorithm for CTL* works in stages such that in stage i formulas of level i are processed. Let f be a CTL* formula. The state subformulas of level i are defined inductively as follows:

- Level 0 contains all atomic propositions.
- Level $i + 1$ contains all state subformulas g such that all state subformulas of g are of level i or less and g is not contained in any lower level.

To illustrate the levels of a CTL* formula we return to the microwave oven example. The CTL* formula given below asserts that whenever an illegal sequence of steps occurs, then either the oven will never heat or it will eventually be reset.

$$\mathbf{AG}((\neg Close \wedge Start) \rightarrow \mathbf{A}(\mathbf{G}\, \neg Heat \vee \mathbf{F}\, \neg Error))$$

The illegal sequence is described by $(\neg Close \wedge Start)$, which means that the start button is pressed before the door is closed. The result of the reset step is indicated by $\neg Error$. This property is not expressible in CTL.

In order to simplify the model checking we consider only existential path quantifiers. Thus, the above formula is rewritten as

$$\neg\, \mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error))$$

The levels of the subformulas of this formula are:

- Level 0 subformulas are $Close$, $Start$, $Heat$, and $Error$;
- level 1 subformulas are $\mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error)$ and $\neg Close$;
- level 2 subformula is $\mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error))$, and
- level 3 contains the entire formula.

Let g be a CTL* formula, then a subformula $\mathbf{E}\, h_1$ of g is *maximal* iff $\mathbf{E}\, h_1$ is not a strict subformula of any strict subformula $\mathbf{E}\, h$ of g. For example, consider the formula

$$\mathbf{E}(a \vee \mathbf{E}(b \wedge \mathbf{EF}\, c)).$$

Then, $\mathbf{EF}\, c$ is a maximal subformula of $\mathbf{E}(b \wedge \mathbf{EF}\, c)$ but not of $\mathbf{E}(a \vee \mathbf{E}(b \wedge \mathbf{EF}\, c))$.

Let $M = (S, R, L)$ be a Kripke structure, let f be a CTL* formula, and let g be a state subformula of f of level i. We assume that the states of M have already been labeled correctly with all state subformulas of level smaller than i. In stage i of the algorithm for CTL*, g is added to all states that make it true. Several cases are considered according to the form of the formula g:

- If g is an atomic proposition, then g is in $label(s)$ iff it is in $L(s)$.
- If $g = \neg g_1$, then g is added to $label(s)$ iff g_1 is not in $label(s)$.
- If $g = g_1 \vee g_2$, then g is added to $label(s)$ iff either g_1 or g_2 are in $label(s)$.

```
procedure CheckE(g)
    if g is a CTL formula then
        apply CTL model checking for g;
        return;
    end if;
    g' := g[a_1/ E h_1, ..., a_k/ E h_k];
    for all s ∈ S
        for i = 1, ..., k do
            if E h_i ∈ label(s) then label(s) := label(s) ∪ {a_i};
    end for all;
    apply LTL model checking for g';
    for all s ∈ S do
        if g' ∈ label(s) then label(s) := label(s) ∪ {g};
    end for all;
end procedure
```

Figure 4.4
Procedure for computing the set of states satisfying the CTL* formula $g = \mathbf{E}\, g_1$.

▪ If $g = \mathbf{E}\, g_1$, then the procedure $CheckE(g)$, given in Figure 4.4, is applied to add g to the label of all states that satisfy the formula, where $\mathbf{E}\, h_1, \ldots, \mathbf{E}\, h_k$ are the maximal subformulas of g, and $a_1, \ldots, a_k$ are fresh atomic propositions. The formula g' in the procedure is obtained by replacing each subformula $\mathbf{E}\, h_i$ by the atomic proposition a_i. Note that, the resulting formula is of the form $\mathbf{E}\, g_1'$ where g_1' is a pure LTL path formula. Here we assume that the LTL model checker updates $label(s)$ so that $M, s \models g'$ if and only if $label(s) := label(s) \cup \{g'\}$.

The complexity of this algorithm depends on the complexity of the model checking algorithms for CTL and LTL that are used. As shown in Section 4.1, the complexity of CTL model checking is linear in both the size of the structure M and the formula f. The best currently known time complexity for LTL model checking is $|M| \cdot 2^{O(|f|)}$.

THEOREM 3 There is a CTL* model checking algorithm with complexity $|M| \cdot 2^{O(|f|)}$.

Note that in an actual implementation there is no need to replace state subformulas by auxiliary atomic propositions. Once the labels of the states are updated with respect to a given subformula, this subformula can be referred to as an atomic proposition.

To demonstrate the CTL* model checking algorithm we consider again the CTL* formula

$$\neg\, \mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error))$$

and check it on the microwave oven model described in Figure 4.3.

At level 0 all atomic propositions are handled. At level 1, the formula $\neg Close$ is first added to the labels of states 1 and 2. The other formula of level 1, $\mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error)$, is a pure LTL formula and therefore is handled by an LTL model checking procedure. Since no state satisfies this formula, it is not added to any state label. At level 2, the formula $\mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error)$ is first replaced by the atomic proposition a. An LTL model-checking procedure is then applied to the pure LTL formula $\mathbf{EF}(\neg Close \wedge Start \wedge a)$. No state is labeled with this formula either and therefore, at level 3 all states are labeled with

$$\neg\, \mathbf{EF}(\neg Close \wedge Start \wedge \mathbf{E}(\mathbf{F}\, Heat \wedge \mathbf{G}\, Error)).$$

Thus, this property always holds for the microwave oven.

5 Binary Decision Diagram

In this chapter we describe how to represent finite state reactive systems symbolically using binary decision diagrams. We first discuss how binary decision diagrams can be used to represent boolean functions. The boolean functions are defined over 0 and 1 where 0 represents *False* and 1 represents *True*. We show that the size of the binary decision diagrams depends strongly on the ordering that is selected for the variables and briefly discuss some heuristics that can be used for selecting good orderings. We also describe how various logical operations can be efficiently implemented using this representation. Next, we explain how to encode Kripke structures using binary decision diagrams, thus enabling both synchronous and asynchronous systems to be represented concisely. This is one of the most important ideas in computer-aided verification in that it makes symbolic model checking possible and permits very large systems to be handled routinely.

5.1 Representing Boolean Formulas

Ordered binary decision diagrams (OBDDs) are a canonical form representation for boolean formulas [34]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of applications in computer aided design, including symbolic simulation, verification of combinational logic and, more recently, verification of finite-state concurrent systems.

To motivate our discussion of binary decision diagrams we first consider *binary decision trees*. A binary decision tree is a rooted, directed tree that consists of two types of vertices, terminal vertices and nonterminal vertices. Each nonterminal vertex v is labeled by a variable $var(v)$ and has two successors: $low(v)$ corresponding to the case where the variable v is assigned 0, and $high(v)$ corresponding to the case where v is assigned 1. Each terminal vertex v is labeled by $value(v)$ which is either 0 or 1. A binary decision tree for the two-bit comparator, given by the formula $f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$, is shown in Figure 5.1. One can decide whether a particular truth assignment to the variables makes the formula true or not by traversing the tree from the root to a terminal vertex. If the variable v is assigned 0, then the next vertex on the path from the root to the terminal vertex will be $low(v)$. If v is assigned 1 then the next vertex on the path will be $high(v)$. The value that labels the terminal vertex will be the value of the function for this assignment. For example, the assignment $\langle a_1 := 1, a_2 := 0, b_1 := 1, b_2 := 1\rangle$ leads to a leaf vertex labeled 0; hence, the formula is false for this assignment.

Binary decision trees do not provide a very concise representation for boolean functions. In fact, they are essentially the same size as truth tables. Fortunately, there is usually a lot of redundancy in such trees. For example, in the tree of Figure 5.1 there are eight subtrees

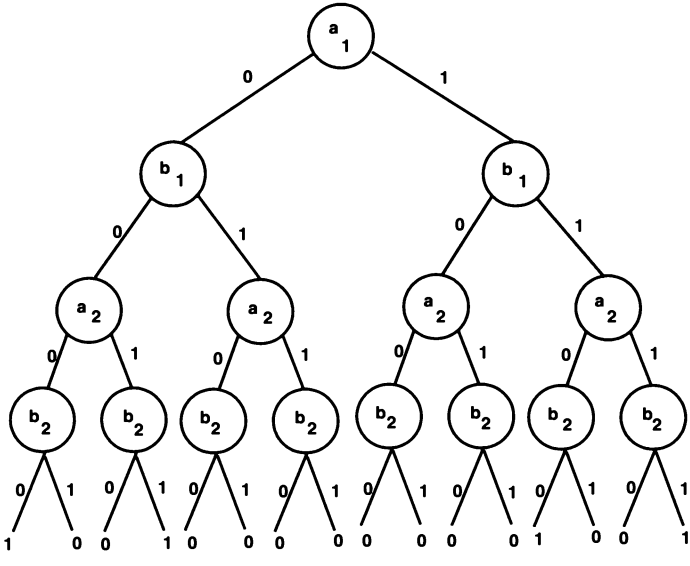


Figure 5.1
Binary decision tree for two-bit comparator.

with roots labeled by b_2, but only three are distinct. Thus, we can obtain a more concise representation for the boolean function by merging isomorphic subtrees. This results in a directed acyclic graph (DAG) called a *binary decision diagram*. More precisely, a binary decision diagram is a rooted, directed acyclic graph with two types of vertices, terminal vertices and nonterminal vertices. As in the case of binary decision trees, each nonterminal vertex v is labeled by a variable $var(v)$ and has two successors, $low(v)$ and $high(v)$. Each terminal vertex is labeled by either 0 or 1. Every binary decision diagram B with root v determines a boolean function $f_v(x_1, \ldots, x_n)$ in the following manner:

1. If v is a terminal vertex:

(a) If $value(v) = 1$ then $f_v(x_1, \ldots, x_n) = 1$.

(b) If $value(v) = 0$ then $f_v(x_1, \ldots, x_n) = 0$.

2. If v is a nonterminal vertex with $var(v) = x_i$ then f_v is the function

$$f_v(x_1, \ldots, x_n) = (\neg x_i \wedge f_{low(v)}(x_1, \ldots, x_n)) \vee (x_i \wedge f_{high(v)}(x_1, \ldots, x_n))$$

In practical applications it is desirable to have a *canonical representation* for boolean functions. Such a representation must have the property that two boolean functions are

logically equivalent if and only if they have isomorphic representations. This property simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not. Two binary decision diagrams are *isomorphic* if there exists a one-to-one and onto function h that maps terminals of one to terminals of the other and nonterminals of one to nonterminals of the other, such that for every terminal vertex v, $value(v) = value(h(v))$ and for every nonterminal vertex v, $var(v) = var(h(v))$, $h(low(v)) = low(h(v))$, and $h(high(v)) = high(h(v))$.

Bryant [34] showed how to obtain a canonical representation for boolean functions by placing two restrictions on binary decision diagrams. First, the variables should appear in the same order along each path from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is achieved by imposing a total ordering $<$ on the variables that label the vertices in the binary decision diagram and requiring that for any vertex u in the diagram, if u has a nonterminal successor v, then $var(u) < var(v)$. The second requirement is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram:

Remove duplicate terminals Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

Remove duplicate nonterminals If two nonterminals u and v have $var(u) = var(v)$, $low(u) = low(v)$ and $high(u) = high(v)$, then eliminate u or v and redirect all incoming arcs to the other vertex.

Remove redundant tests If nonterminal v has $low(v) = high(v)$, then eliminate v and redirect all incoming arcs to $low(v)$.

Starting with a binary decision diagram satisfying the ordering property, the canonical form is obtained by applying the transformation rules until the size of the diagram can no longer be reduced. Bryant shows how this can be done in a bottom-up manner by a procedure called *Reduce* in time which is linear in the size of the original binary decision diagram [34]. The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner. For example, if we use the ordering $a_1 < b_1 < a_2 < b_2$ for the two-bit comparator function, we obtain the OBDD shown in Figure 5.2.

If OBDDs are used as a canonical form for boolean functions, then checking equivalence is reduced to checking isomorphism between binary decision diagrams. Similarly, satisfiability can be determined by checking equivalence to the trivial OBDD that consists of only one terminal labeled by 0.

The size of an OBDD can depend critically on the variable ordering. For example, if we use the variable ordering $a_1 < a_2 < b_1 < b_2$ for the bit-comparator function, we get the OBDD shown in Figure 5.3. Note that this OBDD has eleven vertices, whereas the OBDD

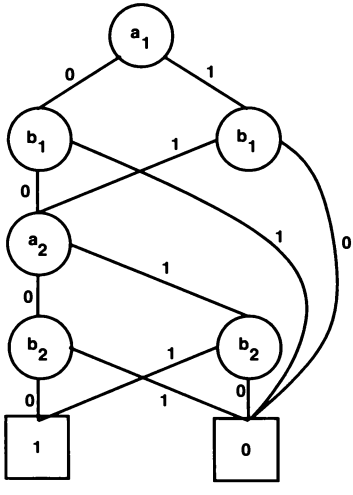


Figure 5.2
OBDD for two-bit comparator.

shown in Figure 5.2 has only eight vertices. In general, for n-bit comparator, if we choose the ordering $a_1 < b_1 < \ldots < a_n < b_n$, then the number of OBDD vertices will be $3n + 2$. On the other hand, if we choose the ordering $a_1 < \ldots < a_n < b_1 \ldots < b_n$, then the number of OBDD vertices is $3 \cdot 2^n - 1$. In general, finding an optimal ordering for the variables is infeasible; in fact, it can be shown that even checking that a particular ordering is optimal is NP-complete [36]. Moreover, there are boolean functions that have exponential size OBDDs for any variable ordering. One example is the boolean function for the middle output (or n-th output) of a combinational circuit to multiply two n bit integers [35, 36].

Several heuristics have been developed for finding a good variable ordering when such an ordering exists. If the boolean function is given by a combinational circuit, then heuristics based on a depth-first traversal of the circuit diagram generally give good results [118, 185]. The intuition for these heuristics comes from the observation that OBDDs tend to be small when related variables are close together in the ordering. The variables appearing in a subcircuit are related in that they determine the subcircuit's output. Hence, these variables should usually be grouped together in the ordering. This may be accomplished by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit diagram. A technique called *dynamic reordering* [225] appears to be useful in those situations where no obvious ordering heuristics apply. When this technique is used, the OBDD package internally reorders the variables periodically in order to reduce the to-

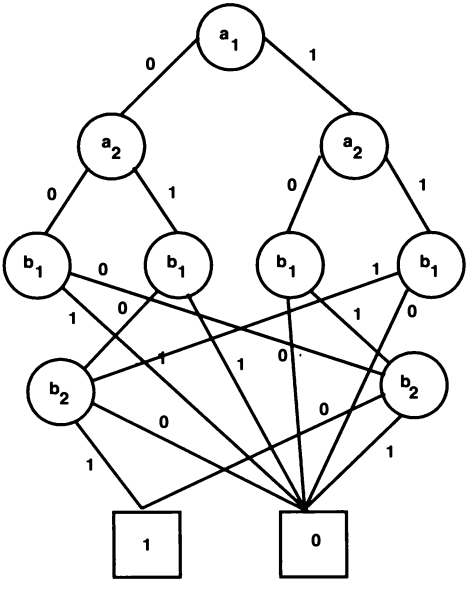


Figure 5.3
OBDD for two-bit comparator.

tal number of vertices in use. The reordering method is designed to save time rather than to find an optimal ordering.

We next explain how to implement various important logical operations using OBDDs. We begin with the function that *restricts* some argument x_i of the boolean function f to a constant value b. This function is denoted by $f|_{x_i \leftarrow b}$ and satisfies the identity

$$f|_{x_i \leftarrow b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If f is represented as an OBDD, then the OBDD for the restriction $f|_{x_i \leftarrow b}$ can be easily computed by a depth-first traversal of the OBDD. For any vertex v which has a pointer to a vertex w such that $var(w) = x_i$, we replace the pointer by $low(w)$ if b is 0 and by $high(w)$ if b is 1. The resulting graph may not be in canonical form, so we apply the *Reduce* function to it in order to obtain the OBDD representation for $f|_{x_i \leftarrow b}$.

All sixteen two-argument logical operations can be implemented efficiently on boolean functions that are represented as OBDDs. In fact, the complexity of these operations is linear in the product of the sizes of the argument OBDDs. The key idea for efficient implementation of these operations is the *Shannon expansion*

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1}).$$

Bryant [34] gives a uniform algorithm called *Apply* for computing all sixteen logical operations. Below we briefly explain how *Apply* works. Let $\star$ be an arbitrary two-argument logical operation, and let f and f' be two boolean functions. To simplify the explanation of the algorithm we introduce the following notation:

- v and v' are the roots of the OBDDs for f and f', and
- $x = var(v)$ and $x' = var(v')$,

We consider several cases depending on the relationship between v and v'.

- If v and v' are both terminal vertices, then $f \star f' = value(v) \star value(v')$.
- If $x = x'$, then we use the Shannon expansion

$$f \star f' = (\neg x \wedge (f|_{x \leftarrow 0} \star f'|_{x \leftarrow 0})) \vee (x \wedge (f|_{x \leftarrow 1} \star f'|_{x \leftarrow 1}))$$

to break the problem into two subproblems. The subproblems are solved recursively. The root of the resulting OBDD will be a new node w with $var(w) = x$, $low(w)$ will be the OBDD for $(f|_{x \leftarrow 0} \star f'|_{x \leftarrow 0})$, and $high(w)$ will be the OBDD for $(f|_{x \leftarrow 1} \star f'|_{x \leftarrow 1})$.

- If $x < x'$, then $f'|_{x \leftarrow 0} = f'|_{x \leftarrow 1} = f'$ since f' does not depend on x. In this case the Shannon expansion simplifies to

$$f \star f' = (\neg x \wedge (f|_{x \leftarrow 0} \star f')) \vee (x \wedge (f|_{x \leftarrow 1} \star f'))$$

and the OBDD for $f \star f'$ is computed recursively as in the second case.

- If $x' < x$, then the required computation is similar to the previous case.

Since each subproblem can generate two subproblems, care must be used in order to prevent the algorithm from being exponential. By using dynamic programming, it is possible to keep the algorithm polynomial. Each subproblem corresponds to a pair of OBDDs that are subgraphs of the original OBDDs for f and f'. Since each subgraph is uniquely determined by its root, the number of subgraphs in the OBDD for f is bounded by the size of the OBDD for f. A similar bound holds for f'. Thus, the number of subproblems is bounded by the product of the size of the OBDDs for f and f'. A hash table, called a *result cache*, is used to record previously computed subproblems. Before any recursive call, the cache is checked to see if the subproblem has been solved. If it has, the result is obtained from the cache; otherwise, the recursive call is performed. The result must be reduced to ensure that it is in canonical form. It is then stored in the result cache.

Boolean negation is one of the sixteen two-argument logical operations that can be implemented using *Apply*. However, it is easier to compute the OBDD for the negation of f simply by negating the value of each terminal node in the OBBD of f.

Several extensions have been developed to decrease the space requirements of Bryant's original OBDD representation for boolean functions [27]. A single multirooted graph can be used to represent a collection of boolean functions that share subgraphs. The same variable ordering is used for all of the formulas in the collection. As in the case of standard OBDDs, the graph contains no isomorphic subgraphs or redundant vertices. If this extension is used then two functions in the collection are identical if and only if they have the same root. Consequently, checking whether two functions are equal can be implemented in constant time. Another useful extension is adding labels to the arcs in the graph to denote boolean negation. This makes it unnecessary to use different subgraphs to represent a formula and its negation. Modern OBDD packages permit graphs with millions of vertices to be manipulated efficiently.

OBDDs can also be viewed as a form of deterministic finite automata [238]. An n-argument boolean function can be identified with the set of strings in $\{0, 1\}^n$ that evaluate to 1. Since this is a finite language and all finite languages are regular, there is a minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original boolean function. Logical operations on boolean functions can be implemented by set operations on the languages accepted by the finite automata. For example, AND corresponds to set intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages. The standard OBDD operations can be viewed as analogs of these constructions.

5.2 Representing Kripke Structures

OBDDs are extremely useful for obtaining concise representations of relations over finite domains [46, 191]. We will see later how to use such representations to describe Kripke structures and to analyze them. If Q is an n-ary relation over $\{0, 1\}$, then Q can be represented by the OBDD for its *characteristic function*

$$f_Q(x_1, \ldots, x_n) = 1 \text{ iff } Q(x_1, \ldots, x_n).$$

Otherwise, let Q be an n-ary relation over the finite domain D. Without loss of generality we assume that D has 2^m elements for some $m > 1$. In order to represent Q as an OBDD, we encode elements of D, using a bijection $\phi : \{0, 1\}^m \rightarrow D$ that maps each boolean vector of length m to an element of D. Using the encoding ϕ, we construct a boolean relation $\hat{Q}$ of arity $m \times n$ according to the following rule:

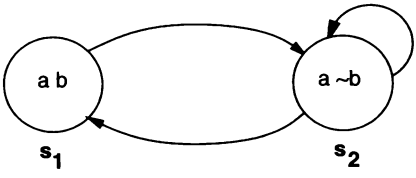


Figure 5.4
Two-state Kripke structure.

$$\hat{Q}(\bar{x}_1, \ldots, \bar{x}_n) = Q(\phi(\bar{x}_1), \ldots, \phi(\bar{x}_n))$$

where $\bar{x}_i$ is a vector of m boolean variables that encodes the variable x_i, which takes values in D. Q can now be represented as the OBBD determined by the characteristic function $f_{\hat{Q}}$ of $\hat{Q}$. This technique can be easily extended to relations over different domains $D_1, \ldots, D_n$. Moreover, because sets can be viewed as unary relations, the same technique can be used to represent sets as OBDDs.

Consider now the Kripke structure $M = (S, R, L)$. To represent this structure, we must describe the set S, the relation R, and the mapping L. For the set S, we first need to encode the states; for simplicity, we assume that there are exactly 2^m states. As above, we let ϕ: $\{0, 1\}^m \rightarrow S$ be a function mapping boolean vectors to states. Since each assignment is the encoding of a state in S, the characteristic function representing S is the OBDD for 1. For the transition relation R, we use the same encoding for the states. As in Chapter 2, we will need two sets of boolean variables, one to represent the starting state and another to represent the final state of a transition. If the transition relation R is encoded by the boolean relation $\hat{R}(\bar{x}, \bar{x}')$, then R is represented by the characteristic function $f_{\hat{R}}$. Finally we consider the mapping L. Although L is defined as a mapping from states to subsets of atomic propositions, it will be more convenient to consider it as a mapping from atomic propositions to subsets of states. The atomic proposition p is mapped to the set of states that satisfy it: $\{ s \mid p \in L(s) \}$. Call this set of states L_p; it can be represented using the encoding ϕ as above. We represent each atomic proposition separately in this way.

In order to illustrate how OBDDs can be used to represent a Kripke structure, consider the two-state structure shown in Figure 5.4. In this case there are two state variables, a and b. We introduce two additional state variables, a' and b', to encode successor states. Thus, we will represent the transition from state s_1 to state s_2 by the conjunction

$$(a \wedge b \wedge a' \wedge \neg b').$$

The boolean formula for the entire transition relation is given by

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b').$$

There are three disjuncts in the formula because the Kripke structure has three transitions. This formula is now converted to an OBDD to obtain a concise representation for the transition relation.

Sometimes we also want to describe sets of possible initial states or fair Kripke structures. The set of initial states is represented in the same way as any other set. For the fairness constraint $F = \{P_1, \ldots, P_n\}$, we simply represent each P_i separately. From now on, we will generally use the same name for a relation, such as R, and for the encoded version of the relation, $\hat{R}$.

In many cases, building an explicit representation of the Kripke structure M and then encoding it as above is not feasible because the structure is too large, even when the final symbolic representation would be concise. Thus, in practice we construct the OBDDs in the representation directly from some concise high-level description of the system. The translation procedure given in Chapter 2 converts systems into formulas. If the domain is encoded as described above, this procedure can be used to construct an OBDD for the transition relation directly from a high-level description of the system.

6 Symbolic Model Checking

In this chapter we describe an efficient algorithm that uses the OBDD representation for Kripke structures to perform model checking. This model checking algorithm is called *symbolic* because it is based on the manipulation of boolean formulas. Because the OBDDs represent sets of states and transitions, we need to operate on entire sets rather than on individual states and transitions. For this purpose, we use a *fixpoint* characterization of the temporal logic operators. A set $S' \subseteq S$ is a fixpoint of a function $\tau: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ if $\tau(S') = S'$. In section 6.1 we show how the set of states satisfying a CTL formula can be characterized as a least or greatest fixpoint of an appropriate function. Iterative techniques based only on set operations are used to calculate these fixpoints. In Section 6.2 we give a CTL model checking algorithm that requires only standard OBDD operations. The incorporation of fairness constraints and the generation of counterexamples is presented in Sections 6.3 and 6.4. In Section 6.5 we show how symbolic model checking can be used to verify a pipelined arithmetic logic unit. Section 6.6 discusses some efficiency issues in symbolic model checking. The chapter concludes with a discussion of how the symbolic CTL model checking algorithm can be applied to LTL model checking and testing language containment.

6.1 Fixpoint Representations

Let $M = (S, R, L)$ be an arbitrary finite Kripke structure. The set $\mathcal{P}(S)$ of all subsets of S forms a lattice under the set inclusion ordering. In this chapter, we will use $\mathcal{P}(S)$ to denote the lattice. Each element S' of the lattice can also be thought of as a *predicate* on S, where the predicate is viewed as being *true* for exactly the states in S'. The least element in the lattice is the empty set, which we also refer to as *False*, and the greatest element in the lattice is the set S, which we sometimes write as *True*. A function that maps $\mathcal{P}(S)$ to $\mathcal{P}(S)$ will be called a *predicate transformer*. Let $\tau: \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ be such a function; then

1. τ is *monotonic* provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$;
2. τ is $\cup$-*continuous* provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau(\cup_i P_i) = \cup_i \tau(P_i)$;
3. τ is $\cap$-*continuous* provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau(\cap_i P_i) = \cap_i \tau(P_i)$.

We write $\tau^i(Z)$ to denote i applications of τ to Z. More formally, $\tau^i(Z)$ is defined recursively by $\tau^0(Z) = Z$ and $\tau^{i+1}(Z) = \tau(\tau^i(Z))$. A monotonic predicate transformer τ on $\mathcal{P}(S)$ always has a least fixpoint, $\mu Z \,.\, \tau(Z)$, and a greatest fixpoint, $\nu Z \,.\, \tau(Z)$ (see Tarski [240]): $\mu Z \,.\, \tau(Z) = \cap\{\, Z \mid \tau(Z) \subseteq Z \,\}$ whenever τ is monotonic, and $\mu Z \,.\, \tau(Z) = \cup_i \tau^i(\mathit{False})$ whenever τ is also $\cup$-continuous. Similarly, $\nu Z \,.\, \tau(Z) = \cup\{\, Z \mid \tau(Z) \supseteq Z \,\}$ whenever τ is monotonic, and $\nu Z \,.\, \tau(Z) = \cap_i \tau^i(\mathit{True})$ whenever τ is also $\cap$-continuous.

The following lemmas are useful in working with predicate transformers defined on finite Kripke structures [61, 107].

```
function Lfp(Tau : PredicateTransformer) : Predicate
    Q := False;
    Q' := Tau(Q);
    while (Q ≠ Q') do
        Q := Q';
        Q' := Tau(Q');
    end while;
    return(Q);
end function
```

Figure 6.1
Procedure for computing least fixpoints.

LEMMA 5 If S is finite and τ is monotonic, then τ is also $\cup$-continuous and $\cap$-continuous.

Proof Let $P_1 \subseteq P_2 \subseteq \ldots$ be a sequence of subsets of S. Because S is finite, there is j_0 such that for every $j \geq j_0$, $P_j = P_{j_0}$. For every $j < j_0$, $P_j \subseteq P_{j_0}$. Thus, $\cup_i P_i = P_{j_0}$ and as a result, $\tau(\cup_i P_i) = \tau(P_{j_0})$. On the other hand, because τ is monotonic, $\tau(P_1) \subseteq \tau(P_2) \subseteq \ldots$. Thus, for every $j < j_0$, $\tau(P_j) \subseteq \tau(P_{j_0})$ and for every $j \geq j_0$, $\tau(P_j) = \tau(P_{j_0})$. As a result, $\cup_i \tau(P_i) = \tau(P_{j_0})$, and τ is $\cup$-continuous. The proof that τ is $\cap$-continuous is similar. □

LEMMA 6 If τ is monotonic, then for every i, $\tau^i(False) \subseteq \tau^{i+1}(False)$ and $\tau^i(True) \supseteq \tau^{i+1}(True)$.

LEMMA 7 If τ is monotonic and S is finite, then there is an integer i_0 such that for every $j \geq i_0$, $\tau^j(False) = \tau^{i_0}(False)$. Similarly, there is some j_0 such that for every $j \geq j_0$, $\tau^j(True) = \tau^{j_0}(True)$.

LEMMA 8 If τ is monotonic and S is finite, then there is an integer i_0 such that $\mu Z \,.\, \tau(Z) = \tau^{i_0}(False)$. Similarly, there is an integer j_0 such that $\nu Z \,.\, \tau(Z) = \tau^{j_0}(True)$.

As a consequence of the preceding lemmas, if τ is monotonic, its least fixpoint can be computed by the program in Figure 6.1.

The invariant for the while loop in the body of the procedure is given by the assertion

$$(Q' = \tau(Q)) \wedge (Q' \subseteq \mu Z \,.\, \tau(Z))$$

It is easy to see that at the beginning of the i-th iteration of the loop, $Q = \tau^{i-1}(False)$ and $Q' = \tau^i(False)$. Lemma 6 implies that

$$False \subseteq \tau(False) \subseteq \tau^2(False) \subseteq \cdots.$$

Consequently, the maximum number of iterations before the while loop terminates is

```
function Gfp(Tau : PredicateTransformer) : Predicate
    Q := True;
    Q' := Tau(Q);
    while (Q ≠ Q') do
        Q := Q';
        Q' := Tau(Q');
    end while;
    return(Q);
end function
```

Figure 6.2
Procedure for computing greatest fixpoints.

bounded by the number of elements in the set S. When the loop does terminate, we will have that $Q = \tau(Q)$ and that $Q \subseteq \mu Z \,.\, \tau(Z)$. Because Q is also a fixpoint, $\mu Z \,.\, \tau(Z) \subseteq Q$, and hence $Q = \mu Z \,.\, \tau(Z)$. Thus the value returned by the procedure is the required least fixpoint. The greatest fixpoint of τ may be computed in a similar manner by the program in Figure 6.2. Essentially the same argument can be used to show that the procedure terminates and that the value it returns is $\nu Z \,.\, \tau(Z)$.

If we identify each CTL formula f with the predicate $\{ s \mid M, s \models f \}$ in $\mathcal{P}(S)$, then each of the basic CTL operators may be characterized as a least or greatest fixpoint of an appropriate predicate transformer [104].

- $\mathbf{AF}\ f_1 = \mu Z \,.\, f_1 \vee \mathbf{AX}\ Z$
- $\mathbf{EF}\ f_1 = \mu Z \,.\, f_1 \vee \mathbf{EX}\ Z$
- $\mathbf{AG}\ f_1 = \nu Z \,.\, f_1 \wedge \mathbf{AX}\ Z$
- $\mathbf{EG}\ f_1 = \nu Z \,.\, f_1 \wedge \mathbf{EX}\ Z$
- $\mathbf{A}[f_1\ \mathbf{U}\ f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge \mathbf{AX}\ Z)$
- $\mathbf{E}[f_1\ \mathbf{U}\ f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge \mathbf{EX}\ Z)$
- $\mathbf{A}[f_1\ \mathbf{R}\ f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee \mathbf{AX}\ Z)$
- $\mathbf{E}[f_1\ \mathbf{R}\ f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee \mathbf{EX}\ Z)$

Intuitively, least fixpoints correspond to eventualities while greatest fixpoints correspond to properties that should hold forever. Thus, $\mathbf{AF}\ f_1$ has a least fixpoint characterization and $\mathbf{EG}\ f_1$ has a greatest fixpoint characterization.

We will only prove the fixpoint characterizations for **EG** and **EU**. The fixpoint characterizations of the remaining CTL operators can be established in a similar manner. Lemmas 9 to 12 below show that $\mathbf{EG}\ f_1 = \nu Z \,.\, f_1 \wedge \mathbf{EX}\ Z$.

LEMMA 9 $\tau(Z) = f_1 \wedge \mathbf{EX}\ Z$ is monotonic.

Proof Let $P_1 \subseteq P_2$. To show that $\tau(P_1) \subseteq \tau(P_2)$, consider some state $s \in \tau(P_1)$. Then $s \models f_1$ and there exists a state s' such that $(s, s') \in R$ and $s' \in P_1$. Because $P_1 \subseteq P_2$, $s' \in P_2$ as well. Thus, $s \in \tau(P_2)$. □

LEMMA 10 Let $\tau(Z) = f_1 \wedge \mathbf{EX}\ Z$ and let $\tau^{i_0}(True)$ be the limit of the sequence $True \supseteq \tau(True) \supseteq \ldots$. For every $s \in S$, if $s \in \tau^{i_0}(True)$ then $s \models f_1$, and there is a state s' such that $(s, s') \in R$ and $s' \in \tau^{i_0}(True)$.

Proof Let $s \in \tau^{i_0}(True)$, Then because $\tau^{i_0}(True)$ is a fixpoint of τ, $\tau^{i_0}(True) = \tau(\tau^{i_0}(True))$. Thus, $s \in \tau(\tau^{i_0}(True))$. By definition of τ we get that $s \models f_1$ and there is a state s', such that $(s, s') \in R$ and $s' \in \tau^{i_0}(True)$. □

LEMMA 11 $\mathbf{EG}\ f_1$ is a fixpoint of the function $\tau(Z) = f_1 \wedge \mathbf{EX}\ Z$.

Proof Suppose $s_0 \models \mathbf{EG}\ f_1$. Then by the definition of $\models$, there is a path $s_0, s_1, \ldots$ in M such that for all k, $s_k \models f_1$. This implies that $s_0 \models f_1$ and $s_1 \models \mathbf{EG}\ f_1$. In other words, $s_0 \models f_1$ and $s_0 \models \mathbf{EX}\ \mathbf{EG}\ f_1$. Thus, $\mathbf{EG}\ f_1 \subseteq f_1 \wedge \mathbf{EX}\ \mathbf{EG}\ f_1$. Similarly, if $s_0 \models f_1 \wedge \mathbf{EX}\ \mathbf{EG}\ f_1$, then $s_0 \models \mathbf{EG}\ f_1$. Consequently, $\mathbf{EG}\ f_1 = f_1 \wedge \mathbf{EX}\ \mathbf{EG}\ f_1$. □

LEMMA 12 $\mathbf{EG}\ f_1$ is the greatest fixpoint of the function

$$\tau(Z) = f_1 \wedge \mathbf{EX}\ Z.$$

Proof Because τ is monotonic, by lemma 5 it is also $\cap$-continuous. Therefore, in order to show that $\mathbf{EG}\ f_1$ is the greatest fixpoint of τ, it is sufficient to prove that $\mathbf{EG}\ f_1 = \cap_i \tau^i(True)$.

We first show that $\mathbf{EG}\ f_1 \subseteq \cap_i \tau^i(True)$. We establish this claim by applying induction on i to show that, for every i, $\mathbf{EG}\ f_1 \subseteq \tau^i(True)$. Clearly, $\mathbf{EG}\ f_1 \subseteq True$. Assume that $\mathbf{EG}\ f_1 \subseteq \tau^n(True)$. Because τ is monotonic, $\tau(\mathbf{EG}\ f_1) \subseteq \tau^{n+1}(True)$. By Lemma 11, $\tau(\mathbf{EG}\ f_1) = \mathbf{EG}\ f_1$. Hence, $\mathbf{EG}\ f_1 \subseteq \tau^{n+1}(True)$.

To show that $\cap_i \tau^i(True)$ is a subset of $\mathbf{EG}\ f_1$, consider some state $s \in \cap_i \tau^i(True)$. This state is included in every $\tau^i(True)$. Hence, it is also in the fixpoint $\tau^{i_0}(True)$. By Lemma 10, s is the start of an infinite sequence of states in which each state is related to the previous one by the relation R. Furthermore, each state in the sequence satisfies f_1. Thus, $s \models \mathbf{EG}\ f_1$. □

LEMMA 13 $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ is the least fixpoint of the function

$$\tau(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\ Z).$$

Proof First we notice that $\tau(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\ Z)$ is monotonic. By Lemma 5, τ is therefore $\cup$-continuous. It is also straightforward to show that $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ is a fixpoint of

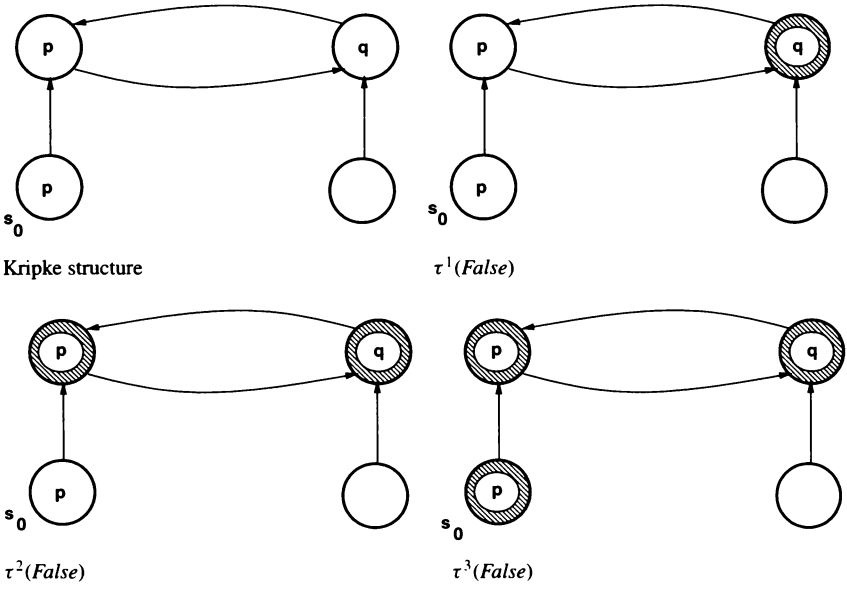


Figure 6.3
Sequence of approximations for $\mathbf{E}[p\ \mathbf{U}\ q]$.

$\tau(Z)$. We still need to prove that $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ is the least fixpoint of $\tau(Z)$. For that, it is sufficient to show that $\mathbf{E}[f_1\ \mathbf{U}\ f_2] = \cup_i \tau^i(False)$. For the first direction, it is easy to prove by induction on i that for every i, $\tau^i(False) \subseteq \mathbf{E}[f_1\ \mathbf{U}\ f_2]$. Consequently, we have that $\cup_i \tau^i(False) \subseteq \mathbf{E}[f_1\ \mathbf{U}\ f_2]$.

The other direction, $\mathbf{E}[f_1\ \mathbf{U}\ f_2] \subseteq \cup_i \tau^i(False)$, is proved by induction on the length of the prefix of the path along which $f_1\ \mathbf{U}\ f_2$ is satisfied. More specifically, if $s \models \mathbf{E}[f_1\ \mathbf{U}\ f_2]$, then there is a path $\pi = s_1, s_2, \ldots$, with $s = s_1$ and $j \geq 1$ such that $s_j \models f_2$ and for all $l < j$, $s_l \models f_1$. We show that for every such state s, $s \in \tau^j(False)$. The basis case is trivial. If $j = 1$, $s \models f_2$ and therefore $s \in \tau(False) = f_2 \vee (f_1 \wedge \mathbf{EX}(False))$.

For the inductive step, assume that the above claim holds for every s and every $j \leq n$. Let s be the start of a path $\pi = s_1, s_2, \ldots$ such that $s_{n+1} \models f_2$ and for every $l < n+1$, $s_l \models f_1$. Consider the state s_2 on the path. It is the start of a prefix of length n along which $f_1 \mathbf{U} f_2$ holds and therefore, by the induction hypothesis, $s_2 \in \tau^n(False)$. Because $(s, s_2) \in R$ and $s \models f_1$, $s \in f_1 \wedge \mathbf{EX}(\tau^n(False))$, thus $s \in \tau^{n+1}(False)$. □

Figure 6.3 shows how the set of states that satisfy $\mathbf{E}[p\ \mathbf{U}\ q]$ may be computed for a simple Kripke structure by using the procedure Lfp. In this case the function τ is given by

$$\tau(Z) = q \vee (p \wedge \mathbf{EX}\ Z).$$

The figure demonstrates how the sequence of approximations $\tau^i(False)$ converges to $\mathbf{E}[p\ \mathbf{U}\ q]$. The states that constitute the current approximation to $\mathbf{E}[p\ \mathbf{U}\ q]$ are shaded. It is easy to see that $\tau^3(False) = \tau^4(False)$. Hence, $\mathbf{E}[p\ \mathbf{U}\ q] = \tau^3(False)$. Because s_0 is in $\tau^3(False)$, we see that $M, s_0 \models \mathbf{E}[p\ \mathbf{U}\ q]$.

6.2 Symbolic Model Checking for CTL

The explicit state model checking algorithm for CTL presented earlier is linear in the size of the graph and in the length of the formula. The algorithm is quite fast in practice [61, 63]. However, an explosion in the size of the model may occur when the state transition graph is extracted from a finite state concurrent system that has many processes or components. In this section, we describe a symbolic model checking algorithm for CTL which operates on Kripke structures. The Kripke structures are represented symbolically using OBDDs, as described in Section 5.2. In order to present the symbolic model checking algorithm, it is convenient to have a more succinct notation for complex operations on boolean formulas. For this, we will use the logic of Quantified Boolean Formulas (QBF) [2, 121].

6.2.1 Quantified Boolean Formulas

Given a set $V = \{v_0, \ldots, v_{n-1}\}$ of propositional variables, QBF(V) is the smallest set of formulas such that

- every variable in V is a formula,
- if f and g are formulas, then $\neg f$, $f \vee g$, and $f \wedge g$ are formulas, and
- if f is a formula and $v \in V$, then $\exists v f$ and $\forall v f$ are formulas.

A *truth assignment* for QBF(V) is a function $\sigma : V \to \{0, 1\}$. If $a \in \{0, 1\}$, then we will use the notation $\sigma\langle v \leftarrow a\rangle$ for the truth assignment defined by

$$\sigma\langle v \leftarrow a\rangle(w) = \begin{cases} a & \text{if } v = w \\ \sigma(w) & \text{otherwise.} \end{cases}$$

If f is a formula in QBF(V) and σ is a truth assignment, we will write $\sigma \models f$ to denote that f is true under the assignment σ. The relation $\models$ is defined inductively in the obvious manner:

- $\sigma \models v$ iff $\sigma(v) = 1$,
- $\sigma \models \neg f$ iff $\sigma \not\models f$,
- $\sigma \models f \vee g$ iff $\sigma \models f$ or $\sigma \models g$,
- $\sigma \models f \wedge g$ iff $\sigma \models f$ and $\sigma \models g$,

- $\sigma \models \exists v f$ iff $\sigma\langle v \leftarrow 0\rangle \models f$ or $\sigma\langle v \leftarrow 1\rangle \models f$, and
- $\sigma \models \forall v f$ iff $\sigma\langle v \leftarrow 0\rangle \models f$ and $\sigma\langle v \leftarrow 1\rangle \models f$.

QBF formulas have the same expressive power as ordinary propositional formulas; however, they are sometimes much more succinct. Every QBF formula determines an n-ary boolean relation on the set V that consists of those truth assignments for the variables in V that make the formula true. We will identify each QBF formula with the boolean relation that it determines. Earlier, we showed how to associate an OBDD with each formula of propositional logic. The quantification operators in QBF can be implemented as combinations of the restrict and apply operators described previously.

- $\exists x f = f|_{x\leftarrow 0} \vee f|_{x\leftarrow 1}$
- $\forall x f = f|_{x\leftarrow 0} \wedge f|_{x\leftarrow 1}$

We will use quantifiers most frequently in *relational product* operations that have the following form

$\exists \bar{v}[f(\bar{v}, \bar{w}) \wedge g(\bar{v}, \bar{x})]$.

6.2.2 The Symbolic Model-Checking Algorithm

The symbolic model-checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be checked as its argument and returns an OBDD that represents exactly those states of the system that satisfy the formula. Of course, the output of *Check* depends on the OBBD representation of the transition relation of the system being checked; this parameter is implicit in the discussion below. We define *Check* inductively over the structure of CTL formulas. If f is an atomic proposition a, then $Check(f)$ is the OBDD representing the set of states satisfying a. If $f = f_1 \wedge f_2$ or $f = \neg f_1$, then $Check(f)$ is obtained by using the function *Apply* described in Section 5.1, with the arguments $Check(f_1)$ and $Check(f_2)$. Formulas of the form $\mathbf{EX}\ f$, $\mathbf{E}[f\ \mathbf{U}\ g]$, and $\mathbf{EG}\ f$ are handled by the procedures:

$Check(\mathbf{EX}\ f) = CheckEX(Check(f))$,

$Check(\mathbf{E}[f\ \mathbf{U}\ g]) = CheckEU(Check(f), Check(g))$,

$Check(\mathbf{EG}\ f) = CheckEG(Check(f))$.

Notice that these intermediate procedures take OBBDs as their arguments, whereas *Check* takes a CTL formula as its argument. The cases of CTL formulas of the form $f \vee g$ or $\neg f$ are handled using the standard algorithms for computing boolean connectives with OBDDs. Because the other temporal operators can all be rewritten using just the ones above, this definition of *Check* covers all CTL formulas.

The procedure for *CheckEX* is straightforward in that the formula **EX** f is true in a state if the state has a successor in which f is true.

$$CheckEX(f(\bar{v})) = \exists \bar{v}' \left[f(\bar{v}') \wedge R(\bar{v}, \bar{v}') \right],$$

where $R(\bar{v}, \bar{v}')$ is the OBBD representation of the transition relation. If we have OBDDs for f and R, then we can compute an OBDD for

$$\exists \bar{v}' \left[f(\bar{v}') \wedge R(\bar{v}, \bar{v}') \right]$$

by using the operations of QBF.

The procedure for *CheckEU* is based on the least fixpoint characterization for the CTL operator **EU** that is given in Section 6.1:

$$\mathbf{E}[f_1 \ \mathbf{U} \ f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge \mathbf{EX}\ Z).$$

We use the function Lfp to compute a sequence of approximations

$$Q_0, Q_1, \ldots, Q_i, \ldots$$

that converges to $\mathbf{E}[f \ \mathbf{U} \ g]$ in a finite number of steps. If we have OBDDs for f, g, and the current approximation Q_i, then we can compute an OBDD for the next approximation Q_{i+1}. Because OBDDs provide a canonical form of boolean functions, it is easy to test for convergence by comparing consecutive approximations. When $Q_i = Q_{i+1}$, the function Lfp terminates. The set of states corresponding to $\mathbf{E}[f \ \mathbf{U} \ g]$ will be represented by the OBDD for Q_i.

CheckEG is similar. In this case the procedure is based on the greatest fixpoint characterization for the CTL operator **EG** that is given in Section 6.1:

$$\mathbf{EG}\ f_1 = \nu Z \,.\, f_1 \wedge \mathbf{EX}\ Z$$

If we have an OBDD for f, then the function Gfp can be used to compute an OBDD representation for the set of states that satisfy **EG** f.

6.3 Fairness in Symbolic Model Checking

Fairness constraints and their significance were discussed in Chapter 3. In Chapter 4, fairness constraints were added to the explicit state model checking algorithm for CTL. In this section we extend the symbolic model checking for CTL, given in the previous section, to include fairness constraints as well. We assume the fairness constraints are given by a set of CTL formulas $F = \{P_1, \ldots, P_n\}$. We define a new procedure *CheckFair* for checking CTL formulas relative to the fairness constraints in F. We do this by defining new inter-

mediate procedures *CheckFairEX*, *CheckFairEU*, and *CheckFairEG*, which correspond to the intermediate procedures used to define *Check*.

Consider the formula **EG** f given fairness constraints F. The formula means that there exists a path beginning with the current state on which f holds globally (invariantly) and each formula in F holds infinitely often on the path. The set of such states Z is the largest set with the following two properties:

1. all of the states in Z satisfy f, and
2. for all fairness constraints $P_k \in F$ and all states $s \in Z$, there is a sequence of states of length one or greater from s to a state in Z satisfying P_k such that all states on the path satisfy f.

This characterization is somewhat different from the one given for the explicit state case in Lemma 1. It is more appropriate for symbolic model checking because it can be expressed by means of a fixpoint as follows:

$$\mathbf{EG}\ f = \nu Z\,.\, f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (Z \wedge P_k)] \tag{6.1}$$

Notice that this formula uses both CTL and fixpoint operators. It is possible to show that this formula is not directly expressible in CTL. In Chapter 7 we will describe a very expressive logic called the μ-calculus, which includes both the least and greatest fixpoint operators. The hybrid formula given above for the fair version of **EG** can be easily translated into the μ-calculus.

Below we prove the correctness of Equation 6.1. We split the proof into two lemmas. The first lemma shows that **EG** f is a fixpoint of the equation

$$Z = f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (Z \wedge P_k)]. \tag{6.2}$$

Thus, it is included in the greatest fixpoint. The second shows that the greatest fixpoint of the equation is included in **EG** f. Combining the two parts of the proof, it follows that **EG** f is the greatest fixpoint.

LEMMA 14 The fair version of **EG** f is a fixpoint of the formula in Equation 6.2.

Proof Let $s \in \mathbf{EG}\ f$, then s is the start of a fair path all of whose states satisfy f. Let s_i be the first state on this path such that $s_i \in P_i$ and $s_i \neq s$. The state s_i is also a start of a fair path along which all states satisfy f. Thus, $s_i \in \mathbf{EG}\ f$. It follows that for every i,

$$s \models f \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (\mathbf{EG}\ f \wedge P_i)]$$

and therefore,

$$s \models f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (\mathbf{EG}\ f \wedge P_k)].$$

Thus, we conclude $\mathbf{EG}\ f \subseteq f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (\mathbf{EG}\ f \wedge P_k)]$.

To show that

$$f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (\mathbf{EG}\ f \wedge P_k)] \subseteq \mathbf{EG}\ f,$$

note that if $s \models f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ (\mathbf{EG}\ f \wedge P_k)]$ then there is a finite path starting from s to a state s' such that $s' \models (\mathbf{EG}\ f \wedge P_k)$. Moreover, every state on the path from s to s' satisfies f, and s' is the beginning of a fair path such that each state on the path satisfies f. Thus, $s \models \mathbf{EG}\ f$, as required. It follows that $\mathbf{EG}\ f$ is a fixpoint. □

LEMMA 15 The greatest fixpoint of the formula in Equation 6.2 is included in $\mathbf{EG}\ f$.

Proof Let Z be an arbitrary fixpoint of the formula in Equation 6.2. We show that Z is included in $\mathbf{EG}\ f$. Assume that $s \in Z$. Then, s satisfies f. Moreover, it has a successor s' that is a start of a path to a state s_1 such that all states on this path satisfy f and s_1 satisfies $Z \wedge P_1$. Because $s_1 \in Z$ we can conclude by the same argument that there is a path from s_1 to a state s_2 in P_2. Using this argument n times we conclude that s is the start of a path along which all the states satisfy f and which passes through $P_1, \ldots, P_n$. Moreover, the last state on this path is in Z. Thus, there is a path from this state back to some state in P_1 and the construction can be repeated.

Induction can be used to show formally that there exists a path starting at s such that f holds on every state on the path and each fairness constraint holds infinitely often. Thus, s is in $\mathbf{EG}\ f$. Because Z is an arbitrary fixpoint, it follows that the greatest fixpoint is contained in $\mathbf{EG}\ f$. □

From the fixpoint characterization it follows that the set of states satisfying $\mathbf{EG}\ f$ under the fairness constraints $F = \{P_1, \ldots, P_n\}$ can be computed by the procedure *CheckFairEG*$(f(\bar{v}))$ according to the following fixpoint characterization:

$$\nu Z(\bar{v}) \,.\, f(\bar{v}) \wedge \bigwedge_{k=1}^{n} \mathbf{EX}(\mathbf{EU}(f(\bar{v}), Z(\bar{v}) \wedge P_k)).$$

The fixpoint can be evaluated in the same manner as before. The main difference is that each time the above expression is evaluated, several nested fixpoint computations are performed (inside *CheckEU*).

Checking **EX** f and **E**$[f$ **U** $g]$ under fairness constraints is similar to the explicit state case. The set of all states which are the start of some fair computation is

$$fair(\bar{v}) = CheckFair(\mathbf{EG}\ True).$$

The formula **EX** f is true under fairness constraints in a state s if and only if there is a successor state s' such that s' satisfies f and s' is at the beginning of some fair computation path. It follows that the formula **EX** f (under fairness constraints) is equivalent to the formula $\mathbf{EX}(f \wedge fair)$ (without fairness constraints). Therefore, we define

$$CheckFairEX(f(\bar{v})) = CheckEX(f(\bar{v}) \wedge fair(\bar{v})).$$

Similarly, the formula **E**$[f$ **U** $g]$ (under fairness constraints) is equivalent to the formula **E**$[f$ **U** $(g \wedge fair)]$ (without fairness constraints). Hence, we define

$$CheckFairEU(f(\bar{v}), g(\bar{v})) = CheckEU(f(\bar{v}), g(\bar{v}) \wedge fair(\bar{v})).$$

6.4 Counterexamples and Witnesses

One of the most important features of CTL model-checking algorithms is the ability to find *counterexamples* and *witnesses*. When this feature is enabled and the model checker determines that a formula with a universal path quantifier is false, it will find a computation path which demonstrates that the negation of the formula is true. Likewise, when the model checker determines that a formula with an existential path quantifier is true, it will find a computation path that demonstrates why the formula is true. For example, if the model checker discovers that the formula **AG** f is false, it will produce a path to a state in which $\neg f$ holds. Similarly, if it discovers that the formula **EF** f is true, it will produce a path to a state in which f holds. Note that the counterexample for a universally quantified formula is the witness for the dual existentially quantified formula. By exploiting this observation we can restrict our discussion of this feature to finding witnesses for the three basic CTL operators **EX**, **EG**, and **EU**.

In order to explain the procedure for finding a witness for some CTL formula we will consider the strongly connected components of the transition graph determined by the Kripke structure. Conceptually, we form a new graph in which the nodes are the strongly connected components and there is an edge from one strongly connected component to another if and only if there is an edge from a state in one to a state in the other. It is easy to see that the new graph does not contain any proper cycles, that is, each cycle in the graph is contained in one of the strongly connected components. Moreover, because we only consider finite Kripke structures, each infinite path must have a suffix that is entirely contained within a strongly connected component of the transition graph.

We start by considering the problem of how to find a witness for the formula **EG** f under the set of fairness constraints $F = \{P_1, \ldots, P_n\}$. We will identify each P_i with the set of states that make it true. Recall that the set of states that satisfy the formula **EG** f with the fairness constraints F is given by the formula

$$\nu Z \,.\, f \wedge \bigwedge_{k=1}^{n} \mathbf{EX}(\mathbf{E}[f \;\mathbf{U}\; Z \wedge P_k]) \tag{6.3}$$

As in the previous section, we will use **EG** f to denote the set of states that satisfy **EG** f under the fairness constraints F. Given a state s in **EG** f, we would like to exhibit a path π starting with s, which satisfies f in every state, and visits every set $P \in F$ infinitely often. We can always find such a path that consists of a finite prefix followed by a repeating cycle. We construct the path incrementally by giving a sequence of prefixes of the path of increasing length until a cycle is found. At each step in the construction we must ensure that the current prefix can be extended to a fair path along which each state satisfies f. This invariant is guaranteed by making sure that each time we add a state to the current prefix, the state satisfies **EG** f.

First, we evaluate the above fixpoint formula. In every iteration of the outer fixpoint computation, we compute a collection of least fixpoints associated with the formulas $\mathbf{E}[f \;\mathbf{U}\; (Z \wedge P)]$, for each fairness constraint $P \in F$. For every constraint P, we obtain an increasing sequence of approximations $Q_0^P \subseteq Q_1^P \subseteq Q_2^P \subseteq \ldots$, where Q_i^P is the set of states from which a state in $Z \wedge P$ can be reached in i or fewer steps, while satisfying f. In the last iteration of the outer fixpoint when $Z = \mathbf{EG}\ f$, we save the sequence of approximations Q_i^P for each P in F.

Now, suppose we are given an initial state s satisfying **EG** f. Then s belongs to the set of states computed in equation 6.3, so it must have a successor in $\mathbf{E}[f \;\mathbf{U}\; (\mathbf{EG}\ f \wedge P)]$ for each $P \in F$. In order to minimize the length of the witness path, we choose the first fairness constraint that can be reached from s. This is accomplished by looking for a successor t of s in the saved sets Q_0^P for all $P \in F$. If no such t is found, we search the sets Q_1^P for all $P \in F$. If we still do not find a suitable t, we search the sets Q_2^P, etc. Because s is in **EG** f, we must eventually find a successor t such that $t \in Q_i^P$. Note that t has a path of length i to a state in $(\mathbf{EG}\ f) \wedge P$ and therefore t is in **EG** f. If $i > 0$, we find a successor of t in Q_{i-1}^P. This is done by finding the set of successors of t, intersecting it with Q_{i-1}^P, and then choosing an arbitrary element of the resulting set. Continuing until $i = 0$, we obtain a path from the initial state s to some state u in $(\mathbf{EG}\ f) \wedge P$. We then eliminate P from further consideration, and repeat the above procedure from u until all of the fairness constraints have been visited. Let s' be the final state of the path obtained thus far.

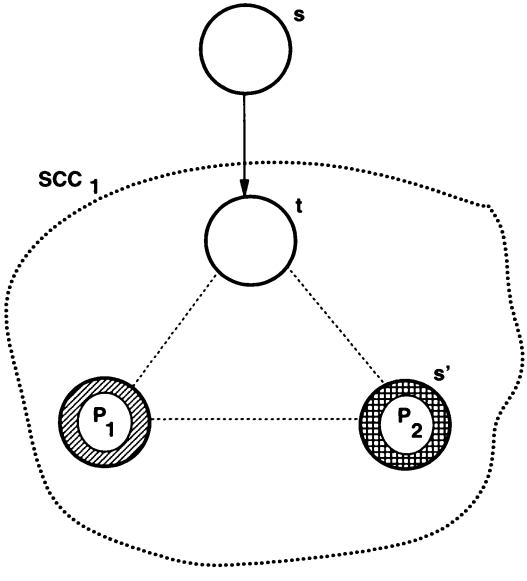


Figure 6.4
Witness is in the first strongly connected component.

To complete a cycle, we need a nontrivial path from s' to the state t along which each state satisfies f. In other words, we need a witness for the formula $\{s'\} \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ \{t\}]$. If this formula is true, we have found the witness path for s. This case is illustrated in Figure 6.4.

If the formula is false, there are several possible strategies. The simplest is to restart the procedure from the final state s' using the entire set of fairness constraints F. Because $\{s'\} \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}\ \{t\}]$ is false, we know that s' is not in the strongly connected component of f containing t; however, s' is in $\mathbf{EG}\ f$. Thus, if we continue this strategy, we must descend in the directed acyclic graph of strongly connected components, eventually either finding a cycle π, or reaching a terminal strongly connected component of f. In the latter case, we are guaranteed to find a cycle, because we cannot exit a terminal strongly connected component. This case is illustrated in Figure 6.5.

A slightly more sophisticated approach would be to precompute $\mathbf{E}[f\ \mathbf{U}\ \{t\}]$. The first time we exit this set, we know the cycle cannot be completed, so we restart from that state. Heuristically, these approaches tend to find short counterexamples (probably because the number of strongly connected components tends to be small), so no attempt is made to find the shortest cycle.

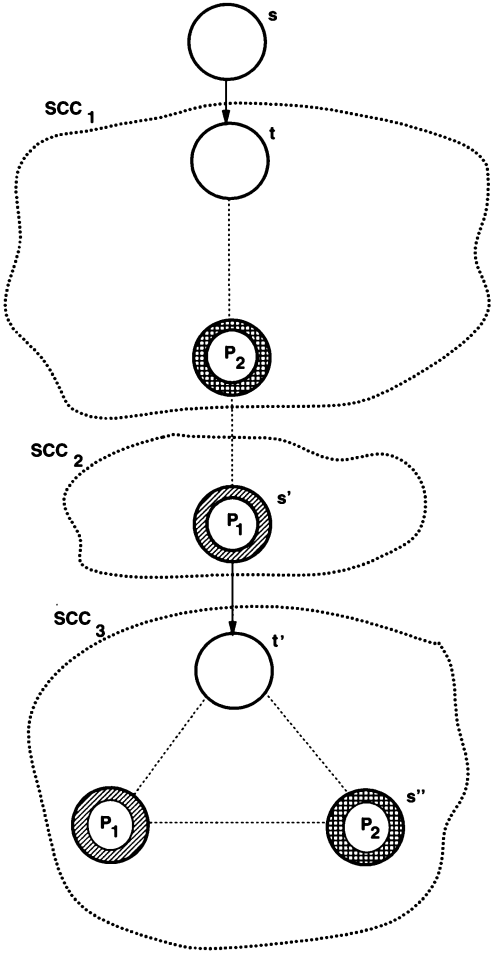


Figure 6.5
Witness spans three strongly connected components.

Finally, we explain how to find witnesses for $\mathbf{E}[f\ \mathbf{U}\ g]$ and $\mathbf{EX}\ f$ in the presence of fairness constraints. Recall that *fair* is the set of states that satisfy $\mathbf{EG}\ True$ under the fairness constraints F. It is possible to compute $\mathbf{E}[f\ \mathbf{U}\ g]$ under F by using the standard CTL model checking algorithm (without fairness constraints) to compute $\mathbf{E}[f\ \mathbf{U}\ (g \wedge fair)]$. Similarly, we can compute $\mathbf{EX}\ f$ by using the standard CTL model checking algorithm to compute $\mathbf{EX}(f \wedge fair)$. The witness procedure for $\mathbf{EG}\ True$ under fairness constraints F can be used to extend witnesses for $\mathbf{E}[f\ \mathbf{U}\ g]$ and $\mathbf{EX}\ f$ to infinite fair paths.

6.5 An ALU Example

This section gives some empirical results to illustrate the effect of using the symbolic representation. We begin by considering a simple pipeline circuit that is specified with CTL.

The pipeline performs three-address arithmetic and logical operations on operands stored in a register file. The circuit is a generalized version of the one described in [45]. Figure 6.6 shows a block diagram for the pipeline. The number of pipe registers can be varied; if r is the number of pipe registers, then executing an instruction requires $r + 2$ cycles.

1. During the first cycle of the instruction, operands are read from the register file into the instruction operand registers.
2. During the second cycle, the result of the operation is computed and stored in the first pipe register.
3. In cycles three through $r + 1$, the result is passed through the remaining pipeline registers. (If there is only one pipe register, these steps are omitted.)
4. In the last cycle, the result is written back to the register file.

The inputs to the circuit specify the operation to be performed and the source and destination registers, but they do not effect the register file until after $r + 2$ cycles. After $r + 2$ cycles, the result of the operation is written back into the register file.

In addition to the address and operation inputs, the pipeline has a *stall* input that indicates that the instruction is invalid and should be ignored. More specifically, the instruction's destination register should not be effected if the *stall* input is true. The *stall* signal might, for example, be used to indicate an instruction cache miss; the signal would be asserted until an instruction is fetched from main memory. In order to allow results to be used before they are actually written into the register file, data can be fed from the ALU output or from one of the pipe registers back to the ALU operand registers. In [45], experiments with a number of different versions of the pipeline are described. Differences caused by varying the number of registers, register widths, number of pipe stages, and number of operations are studied.

The specification of the pipeline is given in CTL. For simplicity of exposition, we give the specification only for a circuit with two general registers and one pipe register, and we assume that the circuit does only XOR operations. In [45] more complex circuits with more operations were verified with respect to the appropriate specifications. The pipeline specification consists of two parts. The first specifies that the destination register in the register file is updated correctly. This is described by a set of formulas of the following form:

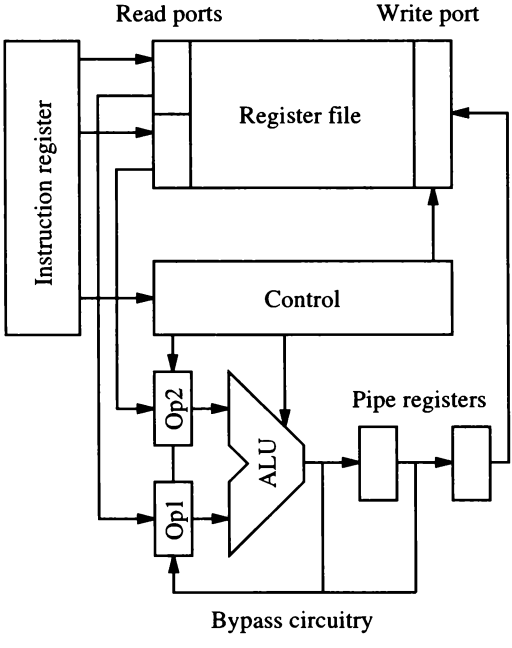


Figure 6.6
Pipeline circuit block diagram.

$$\mathbf{AG}\big(\neg stall \rightarrow ((src1op_i \oplus src2op_i) \equiv result_i)\big).$$

Here, $src1op_i$ and $src2op_i$ are abbreviations for formulas that represent the value of the ith bit of the two source operands and $result_i$ is a formula that represents the ith bit of the result written into the register file. The overall formula states "if the pipeline is not being stalled, then the ith bit of the result of the current operation should be the exclusive-or of the ith bits of the two source operands."

In order to express $src1op_i$, $src2op_i$ and $result_i$, we must account for the latency in the pipeline. For example, the values of the source operands that we need may not have actually been written into the register file when the operation begins. This is because previously issued operations that are still in progress may update one or more of the source registers. However, if we could wait two cycles, any pending updates to the source registers would be completed. Thus, we just need a way of expressing the value stored in a bit of a register some number of cycles in the future. Recall that with one pipe register, the issued operation cannot effect the state of the register file until three cycles in the future. Because of this, the future value can be expressed using the CTL **AX** operator. The value of bit i of register j

in k cycles (where $k \leq 3$) can be found using the CTL expression

$$\underbrace{\mathbf{AX}\,\mathbf{AX}\ldots\mathbf{AX}}_{k}\, reg_{j,i},$$

which we abbreviate as $\mathbf{AX}^k\, reg_{j,i}$. We can check the assumption that the inputs do not effect the register file state before three cycles elapse by verifying that $\mathbf{EX}^k\, reg_{j,i}$ and $\mathbf{AX}^k\, reg_{j,i}$ are equivalent for k up to 3. Now $src1op_i$ is either $\mathbf{AX}^2\, reg_{0,i}$ or $\mathbf{AX}^2\, reg_{1,i}$, depending on whether the first source address is 0 or 1. As discussed, the $\mathbf{AX}^2$ accounts for the pipeline latency; in two cycles, all the values currently being computed will have been written back into the register file. Thus, we obtain

$$src1op_i = (\neg src1addr \wedge \mathbf{AX}^2\, reg_{0,i}) \vee (src1addr \wedge \mathbf{AX}^2\, reg_{1,i}).$$

Here, $src1addr_i$ is the ith bit of the first source address input. The formula for $src2op_i$ is analogous. The formula for $result_i$ is also similar, except we use the values in the register file in three cycles (after the operation is completed), and we select based on *destaddr* the destination address register:

$$result_i = (\neg destaddr \wedge \mathbf{AX}^3\, reg_{0,i}) \vee (destaddr \wedge \mathbf{AX}^3\, reg_{1,i}).$$

The other part of the specification describes what happens to the registers not being written (or to all the registers when the pipeline stalls). In particular, the register should not be altered by the current operation. For example, for register 1:

$$\mathbf{AG}\big((stall \vee \neg destaddr) \rightarrow (\mathbf{AX}^2\, reg_{1,i} \equiv \mathbf{AX}^3\, reg_{1,i})\big).$$

Note that a number of common subformulas, such as the formulas $\mathbf{AX}^k\, reg_{j,i}$, appear throughout the specification. In the experiments described below, the set of states satisfying each of these subformulas was computed only once and then saved.

In one of the experiments, CTL model checking was used to verify an 8-bit wide pipeline with four general registers, one pipe register, and one operation. This example had more than 10^{20} states. The transition relation required about 41,000 OBDD nodes to represent, and the verification required approximately 22 minutes of CPU time on a Sun 3.

These results show a substantial improvement over explicit-state model checking, but it is possible to do even better using the optimizations described in the next section.

6.6 Relational Product Computations

Most of the operations used in the symbolic model checking algorithm are linear in the product of the sizes of the operand OBDDs. The main exception is the relational product

```
function RelProd(f, g : OBDD,  E : setofvariables) : OBDD
    if f = 0 ∨ g = 0 then
        return 0;
    else if f = 1 ∧ g = 1 then
        return 1;
    else if (f, g, E, r) is in the result cache then
        return r;
    else
        let x be the top variable of f;
        let y be the top variable of g;
        let z be the topmost of x and y;
        r0 := RelProd(f|z←0, g|z←0, E);
        r1 := RelProd(f|z←1, g|z←1, E);
        if z ∈ E then
            r := Or(r0, r1);
                /* OBDD for r0 ∨ r1 */
        else
            r := IfThenElse(z, r1, r0);
                /* OBDD for (z ∧ r1) ∨ (¬z ∧ r0) */
        end if;
        insert (f, g, E, r) in the result cache;
        return r;
    end if;
end function
```

Figure 6.7
Relational product algorithm.

operation used to compute **EX** h:

$$\exists \bar{v}'[h(\bar{v}') \wedge R(\bar{v}, \bar{v}')].$$

Although it is possible to implement this operation with one conjunction and a series of existential quantifications, in practice this would be fairly slow. In addition, the OBDD for $h(\bar{v}') \wedge R(\bar{v}, \bar{v}')$ is often much larger than the OBDD for the final result, and we would like to avoid constructing it if possible. For these reasons, we use a special algorithm to compute the OBDD for the relational product in one step from the OBDDs for h and R. Figure 6.7 gives this algorithm for two arbitrary OBDDs f and g.

Like many OBDD algorithms, *RelProd* uses a result cache. In this case, entries in the cache are of the form (f, g, E, r), where E is a set of variables that are quantified out and

f, g and r are OBDDs. If such an entry is in the cache, it means that a previous call to $RelProd(f, g, E)$ returned r as its result.

Although the above algorithm works well in practice, it has exponential complexity in the worst case. Most of the situations where this complexity is observed are cases in which the OBDD for the result is exponentially larger than the OBDDs for the arguments $f(\bar{v})$ and $g(\bar{v})$. In such situations, any method of computing the relational product must have exponential complexity.

6.6.1 Partitioned Transition Relations

The relational product algorithm described previously requires having $R(\bar{v}, \bar{v}')$ as a *monolithic transition relation*, consisting of a single OBDD. We saw in Section 5.2 how to construct this OBDD for synchronous and asynchronous circuits. Unfortunately, for many practical examples, this OBDD is very large. *Partitioned transition relations* can provide a much more concise representation, but they cannot be used with the relational product algorithm given in Figure 6.7. Recall that the transition relations for synchronous and asynchronous circuits have the form of conjunctions or disjunctions of a number of pieces, $R_i(\bar{v}, \bar{v}')$. Each of these pieces can typically be represented by a small OBDD. In our experience, these OBDDs usually have fewer than one hundred nodes, often many fewer; only very rarely do they have more than one thousand nodes. Instead of forming the conjunction or disjunction of the $R_i(\bar{v}, \bar{v}')$ to get $R(\bar{v}, \bar{v}')$, we can represent the circuit by a list of these OBDDs, which are implicitly conjuncted or disjuncted. We call such a list a *partitioned transition relation* [42, 43].

For synchronous circuits, the R_i are of the form

$$R_i(\bar{v}, \bar{v}') = (v_i' \equiv f_i(\bar{v})),$$

where f_i is the function computed by the combinational logic that determines the value of variable v_i. R is the conjunction of the R_i. If the transition relation is instead represented by a list of the R_i, with an implicit conjunction, then we call this a *conjunctive partitioned transition relation*.

For asynchronous circuits, the R_i are of the form:

$$R_i(\bar{v}, \bar{v}') = \big(v_i' \equiv f_i(\bar{v})\big) \wedge \bigwedge_{j \neq i} (v_j' \equiv v_j).$$

The OBDD for R is the disjunction of the R_i. We call the list of the R_i with an implicit disjunction a *disjunctive partitioned transition relation*. In this case the OBDD for R_i can be much larger than the OBDD for f_i (up to a factor of n larger, where n is the number of variables used to encode the state of the circuit). However, there is an additional technique for efficiently representing relations of this form.

Let

$$N_i(\bar{v}, v_i') = v_i' \equiv f_i(\bar{v}).$$

Use the pair $(N_i(\bar{v}, v_i'), i)$ to represent $R_i(\bar{v}, \bar{v}')$ with the interpretation that v_i' is constrained by N_i, and that if $j \neq i$, then v_j' is constrained to be equal to v_j. We exploit this representation during the relational product computation by replacing

$$\exists \bar{v}'[h(\bar{v}') \wedge R_i(\bar{v}, \bar{v}')] = \exists \bar{v}'[h(\bar{v}') \wedge (N_i(\bar{v}, v_i') \wedge \bigwedge_{j \neq i} (v_j' \equiv v_j))]$$

with the equivalent expression

$$\exists v_i'[h(v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n) \wedge N_i(\bar{v}, v_i')].$$

Although a partitioned transition relation with one OBDD for each state variable is often more efficient than constructing a monolithic transition relation, it may not be the best choice. As long as the OBDDs do not become too large, it is better to combine some of the R_i into one OBDD by forming their conjunction or disjunction, as appropriate. Fewer OBDD nodes may be needed in this representation if the R_i that are combined have similar structure near the root of their OBDDs. Combining some of the OBDDs in a partitioned transition can also speed up the relational product computations. Next, we show how to extend the basic algorithm to compute relational products for partitioned transition relations.

Disjunctive Partitioning

For a disjunctive partitioned transition relation, the relational product computed is of the form

$$\exists \bar{v}'\big[h(\bar{v}') \wedge (R_0(\bar{v}, \bar{v}') \vee \cdots \vee R_{n-1}(\bar{v}, \bar{v}'))\big].$$

This relational product can be computed without ever constructing the OBDD for the full transition relation by distributing the existential quantification over the disjunctions:

$$\exists \bar{v}' \left[h(\bar{v}') \wedge R_0(\bar{v}, \bar{v}')\right] \vee \cdots \vee \exists \bar{v}' \left[h(\bar{v}') \wedge R_{n-1}(\bar{v}, \bar{v}')\right].$$

Thus, we are able to reduce the problem of computing the relational product to a series of relational products involving relatively small OBDDs. Much larger asynchronous circuits can be verified using this representation than with a monolithic transition relation.

Conjunctive Partitioning

When using a conjunctive partitioned transition relation, the relational product computed is of the form

$$\exists \bar{v}' \left[h(\bar{v}') \wedge (R_0(\bar{v}, \bar{v}') \wedge \cdots \wedge R_{n-1}(\bar{v}, \bar{v}'))\right]. \tag{6.4}$$

The main difficulty in computing this relational product without building the conjunction is that existential quantification does not distribute over conjunction. The method we now describe overcomes this difficulty.

The technique in [42, 43] is based on two observations. First, circuits exhibit locality, so many of the R_i will depend on only a small number of the variables in $\bar{v}$ and $\bar{v}'$. (In the earlier discussion on extracting transition relations from circuits, there was only one primed variable per R_i, but in Section 6.6.2, we will see that it is sometimes advantageous to combine some of the pieces, giving a dependence on multiple primed variables.) Second, although existential quantification does not distribute over conjunction, subformulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. We will take advantage of these observations by conjuncting the $R_i(\bar{v}, \bar{v}')$ with $h(\bar{v}')$ one at a time and using "early quantification" to eliminate each variable v'_j when none of the remaining $R_i(\bar{v}, \bar{v}')$ depends on v'_j.

Consider the modulo 8 counter described in Section 2.2.1. Recall that

$$R_0(\bar{v}, v'_0) = (v'_0 \Leftrightarrow \neg v_0)$$
$$R_1(\bar{v}, v'_1) = (v'_1 \Leftrightarrow v_0 \oplus v_1)$$
$$R_2(\bar{v}, v'_2) = (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

In this case, the relational product for **EX** h is

$$\exists v'_0\, \exists v'_1\, \exists v'_2 \left[h(\bar{v}') \wedge (R_0(\bar{v}, v'_0) \wedge R_1(\bar{v}, v'_1) \wedge R_2(\bar{v}, v'_2))\right].$$

We can rewrite this as

$$\exists v'_2\, \exists v'_1\, \exists v'_0 \left[((h(\bar{v}') \wedge R_0(\bar{v}, v'_0)) \wedge R_1(\bar{v}, v'_1)) \wedge R_2(\bar{v}, v'_2)\right]. \tag{6.5}$$

The reasons for doing the conjunctions and quantifications in this particular order will become clear momentarily. As mentioned earlier, subformulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. Since $R_2(\bar{v}, v'_2)$ does not depend on v'_0 or v'_1, we can re-express the relational product as:

$$\exists v'_2 \left[\exists v'_1\, \exists v'_0 \left[(h(\bar{v}') \wedge R_0(\bar{v}, v'_0)) \wedge R_1(\bar{v}, v'_1)\right] \wedge R_2(\bar{v}, v'_2)\right].$$

Now since $R_1(\bar{v}, v'_1)$ does not depend on v'_0, we obtain

$$\exists v'_2 \left[\exists v'_1 \left[\exists v'_0 \left[(h(\bar{v}') \wedge R_0(\bar{v}, v'_0)\right] \wedge R_1(\bar{v}, v'_1)\right] \wedge R_2(\bar{v}, v'_2)\right].$$

We can compute this relational product by starting with $h(\bar{v}')$ and at each step combining the previous result with an $R_i(\bar{v}, \bar{v}')$ and quantifying out the appropriate variables. Thus,

we have reduced the problem of computing the full relational product to one of performing a series of smaller relational product-like steps. Notice that the intermediate results may depend both on variables in $\bar{v}$ and variables in $\bar{v}'$.

Now we can explain why we chose the ordering of conjuncts given in Equation 6.5. We wish to order the $R_i(\bar{v}, \bar{v}')$ so that the variables in $\bar{v}'$ can be quantified out as soon as possible and the variables in $\bar{v}$ are added as slowly as possible. This is desirable in that it reduces the number of variables that the intermediate OBDDs depend on and hence can greatly reduce the size of these OBDDs. In this particular example, the variables in $\bar{v}'$ are eliminated one at a time, independent of the ordering of the $R_i(\bar{v}, \bar{v}')$. Thus, the optimum ordering for the $R_i(\bar{v}, \bar{v}')$ is determined by how quickly the variables in $\bar{v}$ are added. For each of the variables v_i in $\bar{v}$, consider the number of R_j that depend on v_i: all three depend on v_0, whereas two depend on v_1, and one depends on v_2. Thus, by dealing with R_0 first, we only introduce one new variable, v_0, while at the same time eliminating v_0'. This explains why we chose to combine $h(\bar{v}')$ and $R_0(\bar{v}, \bar{v}')$ as the first step in the computation. Similarly, $R_1(\bar{v}, \bar{v}')$ was chosen next because it introduces only one new variable, v_1, while v_1' is eliminated.

The previous example involved computing the relational product for **EX** h, that is, we computed the predecessors of a set of states. We also sometimes need to compute the successors of a state set. The relational product in this case is quite similar to that described above. However, instead of quantifying out the next state variables when performing the relational product, we quantify out the present state variables. This change may affect the optimal ordering of the $R_i(\bar{v}, \bar{v}')$ when using conjunctive partitioning. To illustrate this, we consider the modulo 8 counter again. The relational product for a successor computation has the form:

$$\exists v_0\, \exists v_1\, \exists v_2 \left[h(\bar{v}) \wedge (R_0(v_0, \bar{v}') \wedge R_1(v_0, v_1, \bar{v}') \wedge R_2(v_0, v_1, v_2, \bar{v}'))\right].$$

In this case we write the unprimed variables explicitly and leave the primed variables implicit in the relations R_i. Because conjunction is commutative and associative, we can rewrite this as

$$\exists v_0\, \exists v_1\, \exists v_2 \left[((h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')) \wedge R_1(v_0, v_1, \bar{v}')) \wedge R_0(v_0, \bar{v}')\right].$$

Because $R_0(v_0, \bar{v}')$ does not depend on v_1 or v_2, we get

$$\exists v_0 \left[\exists v_1\, \exists v_2 \left[(h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')) \wedge R_1(v_0, v_1, \bar{v}')\right] \wedge R_0(v_0, \bar{v}')\right].$$

Now $R_1(v_0, v_1, \bar{v}')$ does not depend on v_2, so we obtain

$$\exists v_0 \left[\exists v_1 \left[\exists v_2 \left[(h(\bar{v}) \wedge R_2(v_0, v_1, v_2, \bar{v}')\right] \wedge R_1(v_0, v_1, \bar{v}')\right] \wedge R_0(v_0, \bar{v}')\right].$$

In this particular example, the number of new state variables v_i' in the intermediate OBDDs is independent of the ordering of the $R_i(\bar{v}, \bar{v}')$. However, the number of old state variables v_i remaining at each stage depends on the ordering, and is minimized by the ordering given. Note that this ordering is different from the one in Equation 6.5.

The method described above for computing the relational product for the modulo 8 counter can be generalized to an arbitrary conjunctive partitioned transition relation with n state variables, as follows. The user must choose a permutation ρ of $\{0, \ldots, n-1\}$. This permutation determines the order in which the partitions $R_i(\bar{v}, \bar{v}')$ are combined. For each i, let D_i be the set of variables v_i' that $R_i(\bar{v}, \bar{v}')$ depends on. Also, let

$$E_i = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}.$$

Thus, E_i is the set of variables contained in $D_{\rho(i)}$ that are not contained in $D_{\rho(k)}$ for any k larger than i. The E_i are pairwise disjoint and their union contains all the variables. The relational product for **EX** h can be computed as

$$h_1(\bar{v}, \bar{v}') = \exists_{v_j' \in E_0} \left[h(\bar{v}') \wedge R_{\rho(0)}(\bar{v}, \bar{v}')\right]$$

$$h_2(\bar{v}, \bar{v}') = \exists_{v_j' \in E_1} \left[h_1(\bar{v}, \bar{v}') \wedge R_{\rho(1)}(\bar{v}, \bar{v}')\right]$$

$$\vdots$$

$$h_n(\bar{v}) = \exists_{v_j' \in E_{n-1}} \left[h_{n-1}(\bar{v}, \bar{v}') \wedge R_{\rho(n-1)}(\bar{v}, \bar{v}')\right].$$

The result of the relational product is h_n. Note that if some E_i is empty, then

$$h_{i+1}(\bar{v}, \bar{v}') = \left[h_i(\bar{v}, \bar{v}') \wedge R_{\rho(i)}(\bar{v}, \bar{v}')\right]$$

and no existential quantification will be used at this stage. The ordering ρ has a significant impact on how early in the computation state variables can be quantified out. This affects the size of the OBDDs constructed and the efficiency of the verification procedure. Thus, it is important to choose ρ carefully, just as with the OBDD variable ordering.

We search for a good ordering ρ by using a greedy algorithm to find a good ordering on the variables v_i to be eliminated. For each ordering on the variables, there is an obvious ordering on the relations R_i such that when this relation ordering is used, the variables can be eliminated in the order given by the greedy algorithm.

while $(V \neq \phi)$ **do**
    For each $v \in V$ compute the cost of eliminating v;
    Eliminate variable with lowest cost by updating $\mathcal{C}$ and V;
end while;

Figure 6.8
Algorithm for variable elimination.

The algorithm in Figure 6.8 gives the basic greedy technique. We start with the set of variables V to be eliminated and a collection $\mathcal{C}$ of sets where every $D_i \in \mathcal{C}$ is the set of variables on which R_i depends. We then eliminate the variables one at a time by always choosing the variable with the least cost and then updating V and $\mathcal{C}$ appropriately.

All that remains is to determine the cost metric to use. We will consider three different cost measures. To simplify our discussion, we will use R_v to refer to the relation created when eliminating variable v by taking the conjunction of all the R_i that depend on v and then quantifying out v. We will use D_v to refer to the set of variables on which this R_v depends.

Minimum size The cost of eliminating a variable v is simply $|D_v|$. With this cost function, we always try to insure that the new relation we create depends on the fewest number of variables.

Minimum increase The cost of eliminating variable v is

$$|D_v| - \max_{A \in \mathcal{C}, v \in A} |A|$$

which is the difference between the size of D_v and the size of the largest D_i containing v. The intuition here is to try to avoid eliminating variables which would create a large relation from many small relations. In other words, we prefer to make a small increase in the size of an already large relation than to create a new large relation.

Minimum sum The cost of eliminating variable v is

$$\sum_{A \in \mathcal{C}, v \in A} |A|$$

which is simply the sum of the sizes of all the D_i containing v. Because the cost of conjunction depends on the sizes of the arguments, we approximate this cost by the number of variables on which each of the argument R_i depends.

The overall goal is to minimize the size of the largest BDD created during the elimination process. In our abstraction, this translates to finding an ordering that minimizes the size

of the largest set D_v created during the process. Always making a locally optimal choice does not guarantee an optimal solution and there are counterexamples for each of the three cost functions. In fact, the problem of finding an optimal ordering can be shown to be NP-complete. However, the minimum-sum cost function seems to provide the best approximation of the cost of the actual BDD operations and in practice has the best performance.

6.6.2 Recombining Partitions

Earlier, we described how a synchronous circuit could be represented by a set of transition relations $R_i(\bar{v}, \bar{v}')$, each depending on exactly one variable in $\bar{v}'$. We also pointed out that combining some of the R_i together into one OBDD can result in a smaller representation. Combining parts of a transition relation in this way can also significantly speed up the computation of relational products.

For example, consider the case of an n bit counter. With the usual variable ordering, the number of OBDD nodes needed to represent the transition relation is linear in n in both the monolithic and fully partitioned cases. Suppose $h(\bar{v}')$ represents a single state of the counter. Computing the relational product with the fully partitioned representation requires n OBDD operations, each of which has complexity $O(n)$, for a total complexity of $O(n^2)$. On the other hand, if we use the monolithic relation, we perform one operation of complexity $O(n)$, a savings in time of a factor of n. In practice, we can often get a speed-up by combining all of the OBDDs for any given register, without significantly increasing the number of OBDD nodes in the transition relation.

6.6.3 The ALU Example Revisited

This section gives some empirical results to illustrate the effect of using partitioned transition relations. We consider verifying the ALU circuit of Section 6.5 using partitioned transition relations. From the block diagram, we see that the circuit decomposes naturally into pieces. We used this decomposition as a starting point for breaking the transition relation into parts. Some of the parts, such as the register file, were found to require large OBDDs to represent; we broke these into more pieces. We also found that we could combine some of the parts, such as most of the pipe registers, without increasing the number of OBDD nodes required; we did this to decrease overhead. The final decomposition had the following pieces:

1. control logic;
2. the first pipe register;
3. the other pipe registers;

4. the first ALU operand register;
5. the second ALU operand register; and
6. one piece for each general register in the register file.

The ordering above was also the ordering used for processing the transition relation. With this ordering, the number of variables in intermediate results never exceeded the number of state variables by more than the register width. We found that the sizes of the intermediate results with this ordering increased monotonically during each step; thus, breaking the transition relation into pieces did not result in having to manipulate larger state set OBDDs than would have been necessary with a single monolithic OBDD representing the transition relation. This is an important point; in many applications involving OBDDs, it is the number of nodes in intermediate results (not the final result) that limits the size of the problem that can be handled.

We found that verification times grew polynomially in the number of *components* of these example circuits, as we varied the number of registers, the register width, the number of pipeline stages, and the number of pipeline operations. Polynomial verification times were also documented in work [45, 46]. In [23] the authors used symbolic techniques to demonstrate verification times that grow sublinearly in the number of *states* of the system, but still exponentially in the number of components.

Using partitioned transition relations, we verified a 32-bit wide pipeline with eight general registers, two pipe registers, and one operation. This example had 406 state variables resulting in more than 10^{120} reachable states, and the verification took one hour and twenty-five minutes of CPU time on a SPARCstation 1+.

For comparison with the nonpartitioned case, we also ran the example with eight-bit registers. With a partitioned representation, the transition relation needed fewer than 750 nodes, a reduction of more than two orders of magnitude compared to the monolithic case. In addition, the verification needed about one tenth the time.

As another example, we consider the verification of an asynchronous circuit for ensuring mutually exclusive access to a shared resource, due to Martin [189, 96]. The circuit consists of a ring of cells. Each cell communicates with a user of the resource and with its left and right neighbors in the ring. Mutual exclusion is ensured by having a single "token" that is passed around the ring. A cell must have the token before granting access to its user. The distributed mutual exclusion circuit is an example of an asynchronous circuit with complex control and no data path.

We studied how the complexity of reachability analysis varied with the number of cells. We combined the transition relations for the gates making up each individual cell, so the number of elements in the partitioning was equal to the number of cells. The largest circuit that we examined had 16 cells, 256 boolean state variables, and over 10^{16} reachable

states. It took slightly less than thirty minutes of CPU time on a SPARCstation 1+ to find the reachable states. The total number of OBDD nodes needed to represent the transition relation and the state sets both grew linearly with the number of cells.

6.7 Symbolic LTL Model Checking

In this section we show how the model checking problem for linear temporal logic can be solved using symbolic techniques. We give a formal proof of correctness of this technique. Some of the theorems and lemmas are quite technical. When reading this section for the first time, their proofs may be skipped.

Let $\mathbf{A}\ f$ be a linear temporal logic formula. Thus, f is a *restricted path formula* in which the only state subformulas are atomic propositions. We wish to determine all of those states $s \in S$ such that $M, s \models \mathbf{A}\ f$. Since $M, s \models \mathbf{A}\ f$ iff $M, s \models \neg\, \mathbf{E}\, \neg f$, it is sufficient to be able to check the truth of formulas of the form $\mathbf{E}\ f$ where f is a restricted path formula.

In Chapter 4, we described the algorithm of Lichtenstein and Pnueli [173] for this problem, that was linear in the size of the model M and exponential in the length of the formula f. Although their algorithm was linear in the size of the model, it was still impractical for large examples because of the state explosion problem. As in the case of CTL model checking, representing the transition relation as an OBDD enables the procedure to be applied to much larger examples. The exponential complexity of the Lichtenstein-Pnueli algorithm in terms of formula length is caused by a tableau construction that may require exponential space in the size of the formula. Fortunately, the tableau can also be represented by an OBDD. To gain an additional reduction in time and space we use a slight modification of the tableau for LTL formulas [46, 65]. In particular, the modified definition often results in a tableau that has a smaller number of states.

We begin with an informal description of the model checking algorithm. Given a formula $\mathbf{E}\ f$ and a Kripke structure M, we construct a *tableau* T for the path formula f. T is a Kripke structure and includes *every* path that satisfies f. By composing T with M, we find the set of paths that appear in both T and M. A state in M will satisfy $\mathbf{E}\ f$ if and only if it is the start of a path in the composition that satisfies f. The CTL model checking procedure described in Section 6.3 is used to find these states.

We now describe the construction of the tableau T in detail. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $T = (S_T, R_T, L_T)$ with AP_f as its set of atomic propositions. Unlike the algorithm of Lichtenstein and Pnueli, we do not use the full closure of the formula. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively as follows:

- $el(p) = \{p\}$ if $p \in AP_f$.
- $el(\neg g) = el(g)$.
- $el(g \vee h) = el(g) \cup el(h)$.
- $el(\mathbf{X}\, g) = \{\mathbf{X}\, g\} \cup el(g)$.
- $el(g\, \mathbf{U}\, h) = \{\mathbf{X}(g\, \mathbf{U}\, h)\} \cup el(g) \cup el(h)$.

Thus, the set of states S_T of the tableau is $\mathcal{P}(el(f))$. The labeling function L_T is defined so that each state is labeled by the set of atomic propositions contained in the state.

In order to construct the transition relation R_T, we need an additional function sat that associates with each subformula g of f a set of states in S_T. Intuitively, $sat(g)$ will be the set of states that satisfy g.

- $sat(g) = \{\, s \mid g \in s \,\}$ where $g \in el(f)$.
- $sat(\neg g) = \{\, s \mid s \notin sat(g) \,\}$.
- $sat(g \vee h) = sat(g) \cup sat(h)$.
- $sat(g\, \mathbf{U}\, h) = sat(h) \cup \big(sat(g) \cap sat(\mathbf{X}(g\, \mathbf{U}\, h))\big)$.

We want the transition relation to have the property that each elementary formula in a state is true in that state. Clearly, if $\mathbf{X}g$ is in some state s, then all the successors of s should satisfy g. Furthermore, because we are dealing with LTL formulas, if $\mathbf{X}g$ is not in s, then s should satisfy $\neg\mathbf{X}g$. Hence, no successor of s should satisfy g. Thus, we define R_T to be

$$R_T(s, s') = \bigwedge_{\mathbf{X}g \in el(f)} s \in sat(\mathbf{X}\, g) \Leftrightarrow s' \in sat(g).$$

Let $g = (\neg heat)\, \mathbf{U}\, close$ be a specification for the microwave oven example from Chapter 4. Figure 6.9 gives the transition relation R_T for the tableau of the formula $\neg g$. To reduce the number of edges, we connect two states s and s' with a bidirectional arrow if there is an edge from s to s' and also from s' to s. Each subset of $el(g)$ is a state of T. When labeling the states in Figure 6.9 we use h as an abbreviation for $heat$ and c as an abbreviation for $close$. For clarity we also include negations of atomic propositions. Note that $sat(\mathbf{X}g) = \{1, 2, 3, 5\}$ since each of these states contains the formula $\mathbf{X}g$. $sat(g) = \{1, 2, 3, 4, 6\}$ since each of these states either contains $close$ or contains $\neg heat$ and $\mathbf{X}g$. $sat(\neg g) = \{5, 7, 8\}$ is the complement of $sat(g)$. There is a transition from each state in $sat(\mathbf{X}g)$ to each state in $sat(g)$ and from each state in the complement of $sat(\mathbf{X}g)$ to each state in the complement of $sat(g)$. This is because the definition of R_T is a conjunction of "if and only if" conditions.

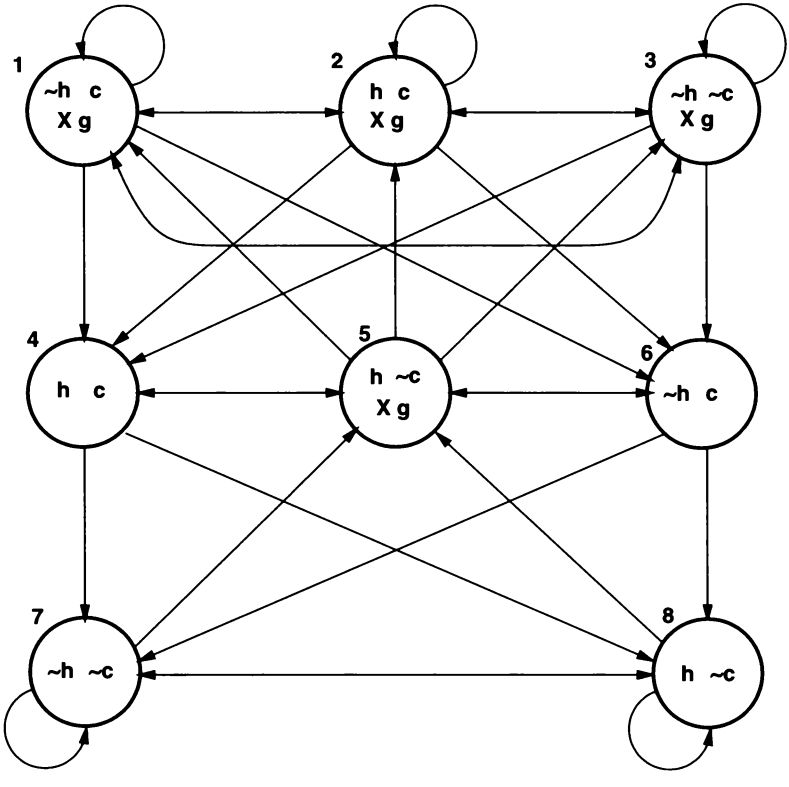


Figure 6.9
Tableau for $(\neg heat)$ **U** $close$.

Unfortunately, the definition of R_T does not guarantee that *eventuality* properties are fulfilled. We can see this behavior in Figure 6.9. Although state 3 belongs to $sat(g)$, the path that loops forever in state 3 does not satisfy the formula g since $close$ never holds on that path. Consequently, an additional condition is necessary in order to identify those paths along which f holds. A path π that starts from a state $s \in sat(f)$ will satisfy f if and only if

- For every subformula g **U** h of f and for every state s on π, if $s \in sat(g$ **U** $h)$ then either $s \in sat(h)$ or there is a later state t on π such that $t \in sat(h)$.

To state the key property of the tableau construction, we must introduce some new notation. Let $\pi' = s'_0, s'_1, \ldots$ be a path in a Kripke structure M, then $label(\pi') =$

$L(s'_0), L(s'_1), \ldots$. Let $l = l_0, l_1, \ldots$ be a sequence of subsets of the set AP and let $AP' \subseteq AP$. The *restriction* of l to AP', denoted by $l\,|_{AP'}$, is the sequence $m_0, m_1, \ldots$ where $m_i = l_i \cap AP'$ for every $i \geq 0$. In addition, we use $sub(f)$ to denote the set of subformulas of f. The following theorem makes precise the intuitive claim that T includes every path which satisfies f.

THEOREM 4 Let T be the tableau for the path formula f. Then, for every Kripke structure M and every path π' of M, if $M, \pi' \models f$ then there is a path π in T that starts in a state in $sat(f)$, such that $label(\pi')\,|_{AP_f} = label(\pi)$.

In order to prove this theorem, we need the following two lemmas. In the remainder of this section, $\pi' = s'_0, s'_1, \ldots$ represents a path in M. We denote the suffix of π' starting from the state s'_i by π'_i, that is, $\pi'_i = s'_i, s'_{i+1}, \ldots$. For the path π'_i, we define

$$s_i = \{\psi | \psi \in el(f) \quad \text{and} \quad M, \pi'_i \models \psi\}. \tag{6.6}$$

Thus, s_i includes all elementary formulas satisfied by the suffix π'_i of π'. Note that s_i is a state in T.

LEMMA 16 For all $g \in sub(f) \cup el(f)$, $M, \pi'_i \models g$ if and only if $s_i \in sat(g)$.

Proof The proof proceeds by induction on the structure of the formula.

1. Let $g \in el(f)$. By the definition of s_i, it is easy to see that $M, \pi'_i \models g$ if and only if $g \in s_i$. By the definition of sat, $g \in s_i$ if and only if $s_i \in sat(g)$. Note that the base case includes all atomic formulas and all formulas of the form $\mathbf{X}\, g$ for any LTL path formula g.

2. Let $g = \neg g_1$ or $g = g_1 \vee g_2$. By the induction hypothesis and the definition of sat, it is easy to prove these cases.

3. Let $g = g_1 \,\mathbf{U}\, g_2$. By the definition of $\mathbf{U}$, $M, \pi'_i \models g_1 \,\mathbf{U}\, g_2$ if and only if $M, \pi'_i \models g_2$ or ($M, \pi'_i \models g_1$ and $M, \pi'_i \models \mathbf{X}(g_1 \,\mathbf{U}\, g_2)$). By the induction hypothesis and the definition of s_i, $M, \pi'_i \models g_2$ or ($M, \pi'_i \models g_1$ and $M, \pi'_i \models \mathbf{X}(g_1 \,\mathbf{U}\, g_2)$) if and only if $s_i \in sat(g_2) \vee (s_i \in sat(g_1) \wedge s_i \in sat(\mathbf{X}(g_1 \,\mathbf{U}\, g_2)))$. Note that $\mathbf{X}(g_1 \,\mathbf{U}\, g_2)$ is in $el(f)$ and therefore has already been handled in the base case. By the definition of sat, $s_i \in sat(g_2) \vee (s_i \in sat(g_1) \wedge s_i \in sat(\mathbf{X}(g_1 \,\mathbf{U}\, g_2)))$ if and only if $s_i \in sat(g_1 \,\mathbf{U}\, g_2)$. □

LEMMA 17 Let $\pi' = s'_0 s'_1 \cdots$ be a path in M. For all $i \geq 0$, let s_i be the tableau state defined by Equation 6.6. Then $\pi = s_0 s_1 \ldots$ is a path in T.

Proof Clearly, for all i, $s_i \in S_T$. By Lemma 16 and the definition of $\mathbf{X}$, it is easy to see the following relation: $s_i \in sat(\mathbf{X}\, g)$ if and only if $M, \pi'_i \models \mathbf{X}\, g$ if and only if $M, \pi'_{i+1} \models g$ if and only if $s_{i+1} \in sat(g)$. By the definition of R_T, if $s_i \in sat(\mathbf{X}\, g) \Leftrightarrow s_{i+1} \in sat(g)$, then $(s_i, s_{i+1}) \in R_T$. Therefore $\pi = s_0 s_1 \ldots$ is a path in T. □

We can now prove Theorem 4.

Proof of Theorem 4 Suppose that, for a path π' in M, $\pi' \models f$. By Lemma 17, we can find a path $\pi = s_0 s_1 \ldots$ in T. By Lemma 16, $s_0 \in sat(f)$. By the definition of s_i given in Equation 6.6, $L(s_i') \mid_{AP_f} = L_T(s_i)$, and thus $label(\pi') \mid_{AP_f} = label(\pi)$. This leads to Theorem 4. □

Next, we want to compute the product $P = (S, R, L)$ of the tableau $T = (S_T, R_T, L_T)$ and the Kripke structure $M = (S_M, R_M, L_M)$.

- $S = \{ (s, s') \mid s \in S_T, s' \in S_M$ and $L_M(s') \mid_{AP_f} = L_T(s) \}$.
- $R((s, s'), (t, t'))$ iff $R_T(s, t)$ and $R_M(s', t')$.
- $L((s, s')) = L_T(s)$.

The transition relation of this product may fail to be total. If this happens, we remove from S all of those states that do not have successors and restrict the transition relation R to the remaining states.

The next lemma states that P contains exactly the sequences π'' for which there are paths π in T and π' in M that have the same labeling of propositions in AP_f.

LEMMA 18 $\pi'' = (s_0, s_0'), (s_1, s_1') \cdots$ is a path in P with $L_P((s_i, s_i')) = L_T(s_i)$ for all $i \geq 0$ if and only if there exist a path $\pi = s_0, s_1 \cdots$ in T, and a path $\pi' = s_0', s_1' \cdots$ in M with $L_T(s_i) = L_M(s_i') \mid_{AP_f}$ for all $i \geq 0$.

The proof of this lemma is straightforward. Given π'' in P, π and π' are obtained by projecting each state on the path onto the appropriate structure. For the other direction, because π and π' agree on the labeling restricted to AP_f, we see that (s_i, s_i') is a state in P for all $i \geq 0$. Moreover, there is a transition from (s_i, s_i') to (s_{i+1}, s_{i+1}').

We extend the function sat to be defined over the set of states of the product P by $(s, s') \in sat(g)$ if and only if $s \in sat(g)$.

We next apply CTL model checking and find the set of all states V in P, $V \subseteq sat(f)$, that satisfy **EG** *true* with the fairness constraints

$$\{ sat(\neg(g \mathbf{U} h) \vee h) \mid g \mathbf{U} h \text{ occurs in } f \}. \tag{6.7}$$

Each of the states in V is in $sat(f)$. Moreover, it is the start of an infinite path that satisfies all of the fairness constraints. These paths have the property that no subformula $g \mathbf{U} h$ holds almost always on the path while h remains false. The correctness of our construction is summarized by the following theorem.

THEOREM 5 $M, s' \models \mathbf{E}\, f$ if and only if there is a state s in T such that $(s, s') \in sat(f)$ and $P, (s, s') \models \mathbf{EG}\, True$ under fairness constraints $\{ sat(\neg(g \mathbf{U} h) \vee h) \mid g \mathbf{U} h$ occurs in $f \}$.

Suppose $M, s' \models \mathbf{E}\, f$ and let π' be a path from s' such that $\pi' \models f$. Let $\pi = s_0, s_1, \ldots$ be a path in the tableau where the individual states s_i are defined by Equation 6.6. The following three lemmas describe properties of π. Lemma 19 proves that this path satisfies the fairness constraints. Lemma 20 shows that if $s \in sat(g_1 \mathbf{U}\, g_2)$ then all of its successors will remain in $sat(g_1 \mathbf{U}\, g_2)$ until a successor in $sat(g_2)$ is reached. Finally, Lemma 21 proves that if a path in the tableau is fair then a necessary and sufficient condition for this path to satisfy f is that its initial state is in $sat(f)$. The last lemma tells us that in order to find paths in the tableau that satisfy f, we should look for fair paths that start in $sat(f)$. This observation extends naturally to the product P.

LEMMA 19 The path π defined above satisfies $\mathbf{G}\, True$ under the fairness constraints given in (6.7).

Proof In order to show that $\pi \models \mathbf{G}\, True$ under the fairness constraints, we need to prove that for every subformula $g \mathbf{U}\, h$ of f, there are infinitely many states s_i on π such that $s_i \in sat(\neg(g \mathbf{U}\, h) \vee h)$. Suppose not, then there exists i_0 such that, for all $i \geq i_0$, $s_i \notin sat(\neg(g \mathbf{U}\, h) \vee h)$. Thus $s_i \in sat(g \mathbf{U}\, h)$ and $s_i \notin sat(h)$. By Lemma 16, for all $i \geq i_0, \pi'_i \models g \mathbf{U}\, h$ and $\pi'_i \not\models h$. Because $\pi'_i \models g \mathbf{U}\, h$ means $\pi'_j \models h$ for some $j \geq i$, this leads to a contradiction. □

LEMMA 20 Assume that for all $k \geq j$, $s_k \in sat(g_1) \Leftrightarrow \pi_k \models g_1$ and $s_k \in sat(g_2) \Leftrightarrow \pi_k \models g_2$. If $\pi_j \not\models g_1 \mathbf{U}\, g_2$ and $s_j \in sat(g_1 \mathbf{U}\, g_2)$, then, for all $k \geq j$, $\pi_k \not\models g_1 \mathbf{U}\, g_2$ and $s_k \in sat(g_1 \mathbf{U}\, g_2)$.

Proof First we prove that, if $s_j \in sat(g_1 \mathbf{U}\, g_2)$ and $\pi_j \not\models g_1 \mathbf{U}\, g_2$, then $s_{j+1} \in sat(g_1 \mathbf{U}\, g_2)$ and $\pi_{j+1} \not\models g_1 \mathbf{U}\, g_2$. From the definition of sat, $s_j \in sat(g_1 \mathbf{U}\, g_2)$ implies $s_j \in sat(g_2)$ or ($s_j \in sat(g_1)$ and $s_j \in sat(\mathbf{X}(g_1 \mathbf{U}\, g_2))$). From the assumptions and the definition of R_T, it follows that:

$$\pi_j \models g_2 \text{ or } (\pi_j \models g_1 \quad \text{and} \quad s_{j+1} \in sat(g_1 \mathbf{U}\, g_2)). \tag{6.8}$$

Because $\pi_j \not\models g_1 \mathbf{U}\, g_2$ implies $\pi_j \not\models g_2$, (6.8) simplifies to

$$\pi_j \models g_1 \text{ and } s_{j+1} \in sat(g_1 \mathbf{U}\, g_2). \tag{6.9}$$

We know that $\pi_j \models g_1$ from (6.9) and $\pi_j \not\models g_1 \mathbf{U}\, g_2$ from the assumption. If π_{j+1} satisfied $g_1 \mathbf{U}\, g_2$, then because $\pi_j \models g_1$ we could conclude that $\pi_j \models g_1 \mathbf{U}\, g_2$. But this is impossible, so it must be the case that $\pi_{j+1} \not\models g_1 \mathbf{U}\, g_2$.

Similarly we can get, for all $k = j+2, j+3, j+4 \ldots$, that $s_k \in sat(g_1 \mathbf{U}\, g_2)$ and $\pi_k \not\models g_1 \mathbf{U}\, g_2$. □

LEMMA 21 Let $\pi \models \mathbf{G}\, True$ under the fairness constraints, then $T, \pi \models f$ if and only if $s_0 \in sat(f)$.

Proof By induction on the structure of the formula, we prove, for each $g \in sub(f) \cup el(f)$ that for all j, $T, \pi_j \models g$ if and only if $s_j \in sat(g)$.

1. Let $g = p \in AP_f$. By the definition of s_j and the definition of sat, it is easy to see the following relation: $\pi_j \models p$ if and only if $p \in L_T(s_j)$ if and only if $p \in s_j$ if and only if $s_j \in sat(p)$.

2. Let $g = \neg g_1$ or $g = g_1 \vee g_2$. By the induction hypothesis and the definition of $\neg$ and $\vee$, it is easy to prove these cases.

3. Let $g = \mathbf{X}\, g_1$. By the definition of R_T and the induction hypothesis, we can see the following relation: $s_j \in sat(\mathbf{X}\, g_1)$ if and only if $s_{j+1} \in sat(g)$ if and only if $\pi_{j+1} \models g$ if and only if $\pi_j \models \mathbf{X}\, g$.

4. Let $g = g_1 \,\mathbf{U}\, g_2$. For the first direction, assume that $\pi_j \models g_1 \,\mathbf{U}\, g_2$, then for some $l \geq j$, $\pi_l \models g_2$ and for all $j \leq i < l$, $\pi_i \models g_1$. By the induction hypothesis, $s_l \in sat(g_2)$ and therefore $s_l \in sat(g_1 \,\mathbf{U}\, g_2)$. By the definition of R_T, it follows that $s_{l-1} \in sat(\mathbf{X}(g_1 \,\mathbf{U}\, g_2))$. But $\pi_{l-1} \models g_1$, so, by induction $s_{l-1} \in sat(g_1)$ and therefore $s_{l-1} \in sat(g_1 \,\mathbf{U}\, g_2)$. By induction on $(l - j)$ we eventually get $s_j \in sat(g_1 \,\mathbf{U}\, g_2)$.

For the other direction, suppose that $s_j \in sat(g_1 \,\mathbf{U}\, g_2)$ and $\pi_j \not\models g_1 \,\mathbf{U}\, g_2$. The inductive hypothesis guarantees that the conditions for Lemma 20 hold. Thus, for all $k \geq j$, $s_k \in sat(g_1 \,\mathbf{U}\, g_2)$ and $\pi_k \not\models g_1 \,\mathbf{U}\, g_2$. This implies that $\pi_k \not\models g_2$, and thus $s_k \notin sat(g_2)$ from the induction hypothesis. Consequently $s_k \in sat(g_1 \,\mathbf{U}\, g_2)$ and $s_k \notin sat(g_2)$ for all $k \geq j$. This leads to a contradiction, because $\pi \models \mathbf{G}\, True$ guarantees that there are infinitely many states s_k such that $s_k \in sat(\neg(g_1 \,\mathbf{U}\, g_2) \vee g_2)$. Therefore if $s_j \in sat(g_1 \,\mathbf{U}\, g_2)$, then $\pi_j \models g_1 \,\mathbf{U}\, g_2$. □

We can now prove Theorem 5.

Proof [of Theorem 5] For the first direction, since $M, s_0' \models \mathbf{E}\, f$, then $\exists \pi' \models f$. By Theorem 4 and Lemma 19, we can prove, for π in T, $\pi \models \mathbf{G}\, True$ and $label(\pi) = label(\pi')\,|_{AP_f}$. By Lemma 18, there is a path π'' in P such that $label(\pi'') = label(\pi)$. Since $label(\pi) = label(\pi')\,|_{AP_f}$ and $\pi' \models f$, we can see that $\pi \models f$. Also since $\pi \models \mathbf{G}\, True$, by Lemma 21 $s_0 \in sat(f)$. Thus $(s_0, s_0') \in sat(f)$. Since $label(\pi) = label(\pi'')$ and $\pi \models \mathbf{G}\, True$, it is clear that $\pi'' \models \mathbf{G}\, True$. Therefore $P, (s_0, s_0') \models \mathbf{EG}\, True$.

For the other direction, because $(s_0, s_0') \in sat(f)$ and $P, (s_0, s_0') \models \mathbf{EG}\, True$, then $\exists \pi'' \models \mathbf{G}\, True$. By Lemma 18, there exist paths $\pi \in T$ and $\pi' \in M$ such that $label(\pi'') = label(\pi) = label(\pi')\,|_{AP_f}$. Since $\pi'' \models \mathbf{G}\, True$ and $label(\pi) = label(\pi'')$, we can see

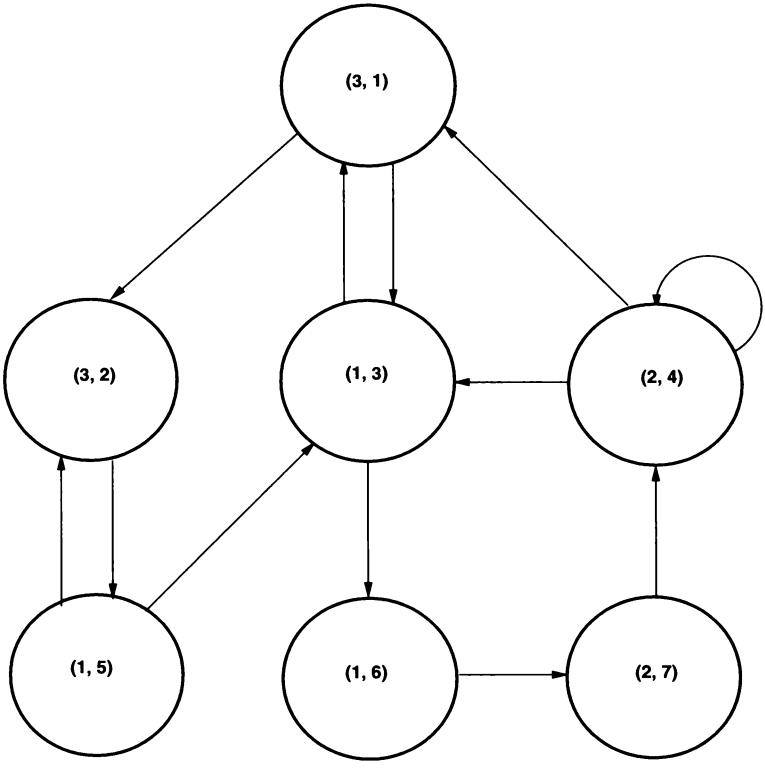


Figure 6.10
The product P of the microwave M and the tableau T.

that $\pi \models \mathbf{G}\, True$. Since $(s_0, s_0') \in sat(f)$, $s_0 \in sat(f)$. From Lemma 21, $\pi \models f$. Since $label(\pi) = label(\pi')\,|_{AP_f}$, $\pi' \models f$ as well. Therefore $M, s_0' \models \mathbf{E}\, f$. □

To illustrate this construction, we check the formula $g = \neg((\neg heat)\ \mathbf{U}\ close)$ on the Kripke structure M in Figure 4.3, describing the microwave oven. The tableau T for this formula is given in Figure 6.9. If we compute the product P as described earlier, we obtain the Kripke structure shown in Figure 6.10. Each state in the product is marked by a pair of states (s', s) where $s' \in T$ and $s \in M$. We have omitted the states (4, 4), (4, 7), (6, 3), (6, 5), (6, 6), (7, 1), and (7, 2) from the diagram for the product structure since they are not the beginning of an infinite path. We use the CTL model checking algorithm to find the set V of states in $sat(\neg g)$ that satisfy the formula $\mathbf{EG}\ true$ with the fairness constraint $sat(\neg((\neg heat)\ \mathbf{U}\ close) \vee close)$. Because $sat(\neg g) = \{(7, 1), (7, 2)\}$ but neither of these

states is the beginning of an infinite path, $V = \emptyset$. We can therefore conclude that no state in M satisfies $\mathbf{E}\ \neg((\neg heat)\ \mathbf{U}\ close)$ and therefore all states satisfy $\mathbf{A}((\neg heat)\ \mathbf{U}\ close)$.

We now describe how the above procedure can be implemented using OBDDs. We assume that the transition relation for M is represented by an OBDD defined over its set of atomic propositions AP. In order to represent the transition relation for T in terms of OBDDs, we associate with each elementary formula g a state variable v_g. If g is an atomic proposition, then v_g is just g itself. Thus, both M and T are defined over the variables in AP_f and some additional state variables.

We describe the transition relation R_T as a boolean formula in terms of two copies $\bar{v}$ and $\bar{v}'$ of the state variables. The boolean formula is converted to an OBDD to obtain a concise representation of the tableau. When the composition P is constructed, it is convenient to separate out the state variables that appear in AP_f. The symbol $\bar{p}$ will be used to denote a boolean vector that assigns truth values to these state variables. Thus, each state in S_T will be represented by a pair $(\bar{p}, \bar{r})$, where $\bar{r}$ is a boolean vector that assigns values to the state variables that appear in the tableau but not in AP_f. A state in S_M will be denoted by a pair $(\bar{p}, \bar{q})$ where $\bar{q}$ is a boolean vector that assigns values to the state variables of M, which are not mentioned in f. Thus, the transition relation R_P for the product of the two Kripke structures will be given by

$$R_P(\bar{p}, \bar{q}, \bar{r}, \bar{p}', \bar{q}', \bar{r}') = R_T(\bar{p}, \bar{r}, \bar{p}', \bar{r}') \wedge R_M(\bar{p}, \bar{q}, \bar{p}', \bar{q}').$$

We use the symbolic model checking algorithm that handles fairness constraints to find the set of states V that satisfy $\mathbf{EG}\ true$ with the fairness constraints given in Equation 6.7. Each state in V is represented by a boolean vector of the form $(\bar{p}, \bar{q}, \bar{r})$. Thus, a state $(\bar{p}, \bar{q})$ in M satisfies $\mathbf{E}\ f$ if and only if there exists $\bar{r}$ such that $(\bar{p}, \bar{q}, \bar{r}) \in V$ and $(\bar{p}, \bar{r}) \in sat(f)$.

7 Model Checking for the μ-Calculus

7.1 Introduction

The propositional μ-calculus is a powerful language for expressing properties of transition systems by using least and greatest fixpoint operators. The μ-calculus has generated much interest among researchers in computer-aided verification. This interest stems from the fact that many temporal and program logics can be encoded into the μ-calculus. Another source of interest in the μ-calculus comes from the existence of efficient model checking algorithms for this formalism. As a consequence, verification procedures for many temporal and modal logics can be described by translating into the μ-calculus. Widespread use of binary decision diagrams has made fixpoint based algorithms even more important because methods that require the manipulation of individual states do not take advantage of this representation.

Several versions of the propositional μ-calculus have been described in the literature, and the algorithms in this chapter will work with any of them. For the sake of concreteness, the propositional μ-calculus of Kozen [157] is used. Closed formulas in this logic evaluate to sets of states. A considerable amount of research has focused on finding techniques for evaluating such formulas efficiently, and many algorithms have been proposed for this purpose. These algorithms generally fall into two categories, local and global.

Local procedures are designed for proving that a specific state of the transition system satisfies the given formula. Because of this, it is not always necessary to examine all the states in the transition system. However, these algorithms have not been combined with BDDs. Tableau-based local approaches have been developed by Cleaveland [75], Stirling and Walker [236], and Winskel [247]. More recently, Andersen [11] and Larsen [169] have developed efficient local methods for a subset of the μ-calculus. Mader [183] has also proposed improvements to the tableau-based method of Stirling and Walker that seem to increase its efficiency.

In this chapter, only global model checking procedures are considered. Global procedures based on BDDs have been shown to be very efficient in practice. These procedures generally work bottom-up through the formula, evaluating each subformula based on the values of its subformulas. Iteration is used to compute the fixpoints. Because of fixpoint nesting, a naive global algorithm may require $O(n^k)$ iterations to evaluate a formula, where n is the number of states in the transition system and k is the depth of nesting of the fixpoints. Emerson and Lei [107] improve on this by observing that successively nested fixpoints of the same type do not increase the complexity of the computation. They formalize this observation using the notion of *alternation depth*. Intuitively, the alternation depth is the number of alternations between least and greatest fixpoint operators in the formula. They give an algorithm that requires only $O(n^d)$ iterations, where d is the alternation depth. Bookkeeping and set manipulations may add another factor of n or so to the time

required. Subsequent work by Andersen, and Cleaveland, Klein, and Steffen [11, 73, 74] has reduced the additional complexity, but the overall number of iterations has remained $O(n^d)$. In [178] this result is improved by giving an algorithm that uses only $O(n^{d/2})$ iterations to compute a formula with alternation depth d. Thus, this algorithm requires only about the square root of the time needed by earlier algorithms.

This chapter describes the propositional μ-calculus and general algorithms for evaluating μ-calculus formulas. Examples of verification problems that can be encoded within the language of the μ-calculus are also provided.

7.2 The Propositional μ-Calculus

Formulas in the μ-calculus are interpreted relative to a transition system. In order to be able to distinguish between different transitions in a system, we modify the definition of a Kripke structure slightly. Instead of having one transition relation R, we will now have a *set* of transition relations T. For simplicity, we will refer to each element α in T as a *transition*, instead of a transition relation. Formally, $M = (S, T, L)$ consists of

- a nonempty set of states S,
- a set of transitions T, such that for each transition $a \in T$, $a \subseteq S \times S$, and
- a mapping $L : S \rightarrow 2^{AP}$ that gives the set of atomic proposition true in a state.

Let $VAR = \{Q, Q_1, Q_2, \ldots\}$ be a set of *relational variables*. Each relational variable $Q \in VAR$ can be assigned a subset of S. The μ-calculus formulas are constructed as follows:

- If $p \in AP$, then p is a formula.
- A relational variable is a formula.
- If f and g are formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are formulas.
- If f is a formula, and $a \in T$, then $[a]f$ and $\langle a \rangle f$ are formulas.
- If $Q \in VAR$ and f is a formula, then $\mu Q \,.\, f$ and $\nu Q \,.\, f$ are formulas, provided that f is *syntactically monotone* in Q, that is, all occurrences of Q within f fall under an even number of negations in f.

Variables in the μ-calculus can be either *free* or *bound* by a fixpoint operator. *Closed formulas* are the formulas without free variables. To emphasize that a formula f contains free relational variables $Q_1, \ldots Q_n$, we sometimes write $f(Q_1, \ldots Q_n)$.

The intuitive meaning of the formula $\langle a \rangle f$ is "it is possible to make an a-transition to a state where f holds." Similarly, $[a]f$ means that "f holds in all states reachable (in

one step) by making an a-transition." The μ and ν operators are used to express least and greatest fixpoints, respectively. The empty set of states is denoted by *False*, and the set of all states S is denoted by *True*. Also, in the rest of this chapter, we will use the more intuitive notation $s \xrightarrow{a} s'$ to mean $(s, s') \in a$.

Formally, a formula f is interpreted as a set of states in which f is true. We write such set of states as $[\![f]\!]_M e$, where M is a transition system and $e : VAR \to 2^S$ is an *environment*. We denote by $e\,[Q \leftarrow W]$ a new environment that is the same as e except that $e\,[Q \leftarrow W]\,(Q) = W$. The set $[\![f]\!]_M e$ is defined recursively as follows.

- $[\![p]\!]_M e = \{s \mid p \in L(s)\}$
- $[\![Q]\!]_M e = e(Q)$
- $[\![\neg f]\!]_M e = S \setminus [\![f]\!]_M e$
- $[\![f \wedge g]\!]_M e = [\![f]\!]_M e \cap [\![g]\!]_M e$
- $[\![f \vee g]\!]_M e = [\![f]\!]_M e \cup [\![g]\!]_M e$
- $[\![\langle a \rangle f]\!]_M e = \{\, s \mid \exists t\, [s \xrightarrow{a} t \text{ and } t \in [\![f]\!]_M e] \,\}$

$[\![[a]\, f]\!]_M e = \{\, s \mid \forall t\, [s \xrightarrow{a} t \text{ implies } t \in [\![f]\!]_M e] \,\}$

- $[\![\mu Q.f]\!]_M e$ is the least fixpoint of the predicate transformer $\tau\colon 2^S \to 2^S$ defined by:

$$\tau(W) = [\![f]\!]_M e\,[Q \leftarrow W]$$

- $[\![\nu Q.f]\!]_M e$ is the greatest fixpoint of the predicate transformer $\tau\colon 2^S \to 2^S$ defined by:

$$\tau(W) = [\![f]\!]_M e\,[Q \leftarrow W]$$

Within formulas, the negation is restricted in use. Thus, monotonicity is guaranteed and the fixpoints are well defined. Formally, every logical connective except negation is monotonic ($f \to f'$ implies $f \wedge g \to f' \wedge g$, $f \vee g \to f' \vee g$, $\langle a \rangle f \to \langle a \rangle f'$, and $[a]f \to [a]f'$), and all the negations can be pushed down to the atomic propositions using De Morgan's laws and the dualities $\neg[a]f \equiv \langle a \rangle \neg f$, $\neg\langle a \rangle f \equiv [a]\neg f$, $\neg\mu Q.f(Q) \equiv \nu Q.\neg f(\neg Q)$, $\neg\nu Q.f(Q) \equiv \mu Q.\neg f(\neg Q)$. Because bound variables are under an even number of negations, they will be negation-free after this process. Thus, each possible formula in a fixpoint operator is monotonic and hence each possible τ is also monotonic ($S \subseteq S'$ implies $\tau(S) \subseteq \tau(S')$). This is enough to ensure the existence of the fixpoints [240]. Furthermore, because we are evaluating formulas over finite transition systems, monotonicity of τ implies that τ is also $\cup$-continuous and $\cap$-continuous (see Lemma 5 in Chapter 6). Hence the least and greatest fixpoints can be computed by iterative evaluation:

$$[\![\mu Q.f]\!]_M e = \bigcup_i \tau^i(\mathit{False}) \qquad [\![\nu Q.f]\!]_M e = \bigcap_i \tau^i(S).$$

where $\tau^i(Q)$ is defined recursively by $\tau^0(Q) = Q$ and $\tau^{i+1}(Q) = \tau(\tau^i(Q))$. Because the domain S is finite, the iteration must stop after a finite number of steps (see Lemma 7 in Chapter 6). More precisely, for some $i, j \leq |S|$, the least fixpoint is equal to $\tau^i(False)$ and the greatest fixpoint is equal to $\tau^j(True)$. To find these fixpoints, we repeatedly apply τ starting from either *False* or *True* until the result does not change.

The *alternation depth* [107] of a formula is the number of alternations in the nesting of least and greatest fixpoints, when all negations are applied only to propositions. In order to make this definition formal, we need to define the top-level ν and μ-subformulas of a μ-calculus formula. A *top-level ν-subformula* of f is a subformula $\nu Q' . g$ of f that is not contained within any other greatest fixpoint subformula of f. For example, the top-level ν-subformulas of $f = \mu Q' . (\nu Q_1 . g_1 \vee \nu Q_2 . g_2)$ are $\nu Q_1 . g_1$ and $\nu Q_2 . g_2$. A *top-level μ-subformula* of f is defined in a similar manner. Formally, the *alternation depth* is defined as follows:

- The alternation depth of an atomic proposition or a relational variable is 0;
- The alternation depth for formulas like $f \wedge g$, $f \vee g$, $\langle a \rangle f$, and $[a]f$ is the maximum alternation depth of the subformulas f and g.
- The alternation depth of $\mu Q . f$ is the maximum of:

1. the constant 1,
2. the alternation depth of f, and
3. one plus the maximum alternation depth of any top-level ν-subformulas of f.

- The *alternation depth* of $\nu Q . f$ is defined symmetrically.

For example, consider a transition system in which $T = \{a\}$. Recall that **EG** f with fairness constraint h holds at a state if there exists a path from the state along which f holds continuously, and h holds infinitely often on this path. This property is expressed using the fixpoint formula (see 6.1)

$$\mathbf{EG}\ f = \nu Z . f \wedge \mathbf{EX}(\mathbf{E}[f\ \mathbf{U}\ (Z \wedge h)]). \tag{7.1}$$

Using the fixpoint characterization of **EU**, we obtain

$$\mathbf{E}[f\ \mathbf{U}\ (Z \wedge h)] = \mu Y . (Z \wedge h) \vee (f \wedge \mathbf{EX}\ Y). \tag{7.2}$$

Substituting the right-hand side of 7.2 in 7.1 gives

$$\nu Z.(f \wedge \mathbf{EX}(\mu Y.(Z \wedge h) \vee (f \wedge \mathbf{EX}\ Y))).$$

Finally, replacing **EX** by $\langle a \rangle$, we obtain the μ-calculus formula

$$\nu Z.(f \wedge \langle a \rangle(\mu Y.(Z \wedge h) \vee (f \wedge \langle a \rangle Y))). \tag{7.3}$$

This formula has an alternation depth of two.

Because of the duality,

$$\nu Q \,.\, f(\cdots, Q, \cdots) = \neg \mu Q \,.\, \neg f(\cdots, \neg Q, \cdots)$$

we could have defined the propositional μ-calculus with just the least fixpoint operator and negation. In order to give a succinct description of certain constructions we sometimes use the dual formulation. However, the concept of alternation depth is easier to define using the formulation given earlier.

7.3 Evaluating Fixpoint Formulas

In this section we give a model checking algorithm for the μ-calculus. This algorithm finds the set of states in a model that satisfy a formula of this logic. Figure 7.1 presents the naive, straightforward, recursive algorithm for evaluating μ-calculus formulas. The time complexity of the algorithm in Figure 7.1 is exponential in the length of the formula. To see this, we analyze the behavior of the algorithm when computing nested fixpoints. The algorithm computes fixpoints by iteratively computing approximations. These successive approximations form a chain of sets ordered by inclusion. Because the number of strict inclusions in such a chain is limited by the number of possible states, it follows that the loop (either in lines 14–17 for a least fixpoint or lines 22–25 for a greatest fixpoint) will execute at most $n + 1$ times, where $n = |S|$. Each iteration of the loop involves a recursive call to evaluate the body of the fixpoint with a different value for the fixpoint variable. If in turn the subformula being evaluated contains a fixpoint, the evaluation of its body will also involve a loop containing up to $n + 1$ recursive calls with a shorter subformula. In general, the body of the innermost fixpoint will be evaluated $O(n^k)$ times where k is the maximum nesting depth of fixpoint operators in the formula.

Note that we have only considered the number of iterations required when evaluating fixpoints and not the number of steps required to evaluate a μ-calculus formula. Although each fixpoint may only take $O(n)$ iterations, each individual iteration can take up to $O(|M| \cdot |f|)$ steps, where $M = (S, T, L)$ is the model and $|M| = |S| + \sum_{a \in T} |a|$. In general, then, this algorithm has time complexity $O[|M| \cdot |f| \cdot n^k]$.

A result by Emerson and Lei [107] demonstrates that the value of a fixpoint formula can be computed with $O((|f| \cdot n)^d)$ iterations, where d is the alternation depth of f. Their algorithm is similar to the straightforward one described above, except when a fixpoint is nested directly within the scope of another fixpoint of the same type. In this case, the fixpoints are computed differently. The basic idea exploits sequences of fixpoints that

```
function eval(f, e)

if f = p then return {s | p ∈ L(s)};
if f = Q then return e(Q);
if f = g1 ∧ g2 then
        return eval(g1, e) ∩ eval(g2, e);
if f = g1 ∨ g2 then
        return eval(g1, e) ∪ eval(g2, e);
if f = ⟨a⟩g then
        return { s | ∃t [s -a-> t and t ∈ eval(g, e)] };
if f = [a]g then
        return { s | ∀t [s -a-> t implies t ∈ eval(g, e)] };

if f = μQ.g(Q) then
        Q_val := False;
        repeat
                Q_old := Q_val;
                Q_val := eval(g, e [Q ← Q_val]);
        until Q_val = Q_old;
        return Q_val;
end if;

if f = νQ.g(Q) then
        Q_val := True;
        repeat
                Q_old := Q_val;
                Q_val := eval(g, e [Q ← Q_val]);
        until Q_val = Q_old;
        return Q_val;
end if;
end function
```

Figure 7.1
Pseudocode for the naive algorithm.

have the same type to reduce the complexity of the algorithm. Then, it is unnecessary to reinitialize computations of inner fixpoints with *False* (for least fixpoint) or *True* (for greatest fixpoint) when calculating new approximations for the outer fixpoints.

A simple example will suffice to demonstrate the idea. When discussing the evaluation of fixpoint formulas, we will use $Q_1, \ldots, Q_k$ as the fixpoint variables, with Q_1 being the outermost fixpoint variable and Q_k being the innermost. We will use the notation $Q_j^{i_1 \cdots i_j}$ to denote the value of the i_j-th approximation for Q_j after having computed the i_l-th approximation for Q_l for $1 \leq l < j$. We use $i_j = \omega$ to indicate that we are considering the final approximation (the actual fixpoint value) for Q_j. For example, Q_1^{ω} is the value of the fixpoint for Q_1 and $Q_2^{3\,0}$ is the initial approximation for Q_2 after having computed the third approximation for Q_1. Consider the formula

$$\mu Q_1 . g_1(Q_1, \mu Q_2 . g_2(Q_1, Q_2)).$$

The subformula $\mu Q_2 \,.\, g_2(Q_1, Q_2)$ defines a monotonic predicate transformer τ taking one set (the value of Q_1) to another set (the value of the least fixpoint of Q_2), that is,

$$\tau(Q_1) = \mu Q_2 \,.\, g_2(Q_1,\ Q_2).$$

When evaluating the outer fixpoint, we start with the initial approximation $Q_1^0 = \mathit{False}$ and then compute $\tau(Q_1^0)$. This is done by iteratively computing approximations for the inner fixpoint also starting from $Q_2^{0\,0} = \mathit{False}$ until we reach a fixpoint $Q_2^{0\,\omega}$. Now Q_1 is increased to Q_1^1, the result of evaluating $g_1(Q_1^0, Q_2^{0\,\omega})$:

$$Q_1^1 = g_1(Q_1^0, Q_2^{0\,\omega}).$$

We next compute the least fixpoint $\tau(Q_1^1)$. Since $Q_1^0 \subseteq Q_1^1$, by monotonicity we know that $\tau(Q_1^0) \subseteq \tau(Q_1^1)$. Note that because τ is monotonic and S is finite, τ is $\bigcup$-continuous. Thus, it is easy to prove by induction that the following lemma holds:

LEMMA 22 If $W \subseteq \bigcup_i \tau^i(\mathit{False})$ then $\bigcup_i \tau^i(W) = \bigcup_i \tau^i(\mathit{False})$.

In other words, to compute a least fixpoint, it is enough to start iterating with any approximation known to be below the fixpoint.

Thus, we can start iterating with $Q_2^{1\,0} = Q_2^{0\omega} = \tau(Q_1^0)$ instead of $Q_2^{1\,0} = \mathit{False}$. When we compute the fixpoint $Q_2^{1\omega}$, we next compute the new approximation to Q_1, which is Q_1^2, the result of evaluating $g_1(Q_1^1, Q_2^{1\omega})$.

$$Q_1^2 = g_1(Q_1^1, Q_2^{1\omega}).$$

Again, we know that $Q_1^1 \subseteq Q_1^2$, which implies that $\tau(Q_1^1) \subseteq \tau(Q_1^2)$. But $\tau(Q_1^1) = Q_2^{1\omega}$, the value of the last inner fixpoint computed, and $\tau(Q_1^2) = Q_2^{2\omega}$ the fixpoint to be computed

next. Again, we can start iterating with any approximation below the fixpoint. So to compute $Q_2^{2\omega}$ we begin with $Q_2^{20} = Q_2^{1\omega} = \tau(Q_1^1)$. In general, when computing $Q_2^{i\omega}$ we always begin with $Q_2^{i0} = Q_2^{(i-1)\omega}$. Because we never restart the inner fixpoint computation, we can have at most n increases in the value of the inner fixpoint variable. Overall, we only need $O(n)$ iterations to evaluate this expression, instead of $O(n^2)$. In general, this type of simplification leads to an algorithm that computes fixpoint formulas in time exponential in the alternation depth of the formula since we only reset an inner fixpoint computation when there is an alternation in fixpoints in the formula.

Assume that the formula f has N fixpoint subformulas. The algorithm uses an array $A[1..N]$ to store the approximations to the fixpoints. Initially, $A[i]$ is set to *False* if the ith fixpoint formula is a least fixpoint and to *True* otherwise. The pseudocode for this algorithm is given in Figure 7.2. When the main operator of the subformula is not a least or greatest fixpoint, the algorithm is the same as the naive algorithm. Unlike the naive algorithm, the approximation values $A[i]$ are not reset when evaluating the subformula $\mu Q_i \,.\, g(Q_i)$ ($\nu Q_i \,.\, g(Q_i)$). Instead, we reset all top-level greatest (least) fixpoint variables contained in g to *True* (*False*). This guarantees that when we evaluate a top-level fixpoint subformula of the same type, we do not start the computation from *False* or *True*, but from the previously computed value as in our example.

In order to understand why the number of iterations of this algorithm is $O((|f| \cdot n)^d)$, note first that the size of the formula $|f|$ is an upper bound on the number of consecutive fixpoints of the same type in f. Because we never reinitialize the computation of inner fixpoints when calculating new approximations for outer fixpoints of the same type, the number of iterations for each such sequence is $O(|f| \cdot n)$ instead of $n^{|f|}$ as in the naive case. The computation is reinitialized at the boundary between two sequences of different types. Thus, with d alternating sequences we have $O((|f| \cdot n)^d)$ iterations altogether.

In [178] formulas with strict alternation of least and greatest fixpoint operators are considered. It is shown there that by storing even more intermediate values, the time complexity for evaluating fixpoint formulas can be reduced to $O(n^{\lfloor d/2 \rfloor + 1})$ where d is again the alternation depth of the formula and $|f|$ is replaced by 1.

7.4 Representing μ-Calculus Formulas Using OBDDs

In this section we describe how to use OBDDs in the model checking algorithms described earlier. First, we show how to encode a transition system $M = (S, T, L)$ into OBDDs. This encoding is similar to the encoding of Kripke structures presented in Section 5.2. The domain S is encoded by the set of values of the n boolean variables $x_1, \cdots, x_n$, that is, S is now the space of boolean vectors of length n. Each variable x_i has a corresponding primed

```
function eval(f, e)

if f = p then return {s | p ∈ L(s)};
if f = Q then return e(Q);
if f = g1 ∧ g2 then
        return eval(g1, e) ∩ eval(g2, e);
if f = g1 ∨ g2 then
        return eval(g1, e) ∪ eval(g2, e);
if f = ⟨a⟩g then
        return { s | ∃t [s -a-> t and t ∈ eval(g, e)] };
if f = [a]g then
        return { s | ∀t [s -a-> t implies t ∈ eval(g, e)] };

if f = μQi.g(Qi) then
        forall top-level greatest fixpoint subformulas νQj.g'(Qj) of g
                do A[j] := True;
        repeat
                Qold := A[i];
                A[i] := eval(g, e [Qi ← A[i]]);
        until A[i] = Qold;
        return A[i];
end if;

if f = νQi.g(Qi) then
        forall top-level least fixpoint subformulas μQj.g'(Qj) of g
                do A[j] := False;
        repeat
                Qold := A[i];
                A[i] := eval(g, e [Qi ← A[i]]);
        until A[i] = Qold;
        return A[i];
end if;
end function
```

Figure 7.2
Pseudocode for the Emerson and Lei algorithm.

variable x_i'. Instead of writing $x_1, \cdots, x_n$, we sometimes use the vector notation $\vec{x}$. Given an interpretation we build the OBDDs corresponding to closed μ-calculus formulas in the following manner.

- Each atomic proposition p has an OBDD associated with it. We will denote this OBDD by $\text{OBDD}_p(\vec{x})$. $\text{OBDD}_p(\vec{x})$ has the property that $\vec{y} \in \{0, 1\}^n$ satisfies OBDD_p if and only if $\vec{y} \in L(p)$.
- Each transition a has an ordered binary decision diagram $\text{OBDD}_a(\vec{x}, \vec{x}')$ associated with it. A boolean vector $(\vec{y}, \vec{z}) \in \{0, 1\}^{2n}$ satisfies OBDD_a if and only if

$(\vec{y}, \vec{z}) \in a$

Now we describe the translation of formulas into OBDDs. Assume that we are given a μ-calculus formula f with free relational variables $Q_1, \cdots, Q_k$. The function **assoc**$[Q_i]$ gives the OBDD corresponding to the set of states associated with the relational variable Q_i. **assoc**$\langle Q \leftarrow B_Q \rangle$ creates a new association by adding a relational variable Q and associating an OBDD B_Q with Q. In other words, **assoc** can be considered as an environment with OBDD representation. The procedure B given below takes a μ-calculus formula f and an association list **assoc** (**assoc** assigns an OBDD to each free relational variable occurring in f) and returns an OBDD corresponding to the semantics of f.

- $B(p, \textbf{assoc}) = \text{OBDD}_p(\vec{x})$.
- $B(Q_i, \textbf{assoc}) = \textbf{assoc}[Q_i]$.
- $B(\neg f, \textbf{assoc}) = \neg B(f, \textbf{assoc})$
- $B(f \wedge g, \textbf{assoc}) = B(f, \textbf{assoc}) \wedge B(g, \textbf{assoc})$.
- $B(f \vee g, \textbf{assoc}) = B(f, \textbf{assoc}) \vee B(g, \textbf{assoc})$.
- $B(\langle a \rangle f, \textbf{assoc}) = \exists \vec{x}'(\text{OBDD}_a(\vec{x}, \vec{x}') \wedge B(f, \textbf{assoc})(\vec{x}'))$, where $B(f, \textbf{assoc})(\vec{x}')$ is the OBBD in which each boolean variable x_i is replaced by its primed version x_i'.
- $B([a]f, \textbf{assoc}) = B(\neg\langle a \rangle \neg f, \textbf{assoc})$.

This equation uses the dual formulation for $[a]$.

- $B(\mu Q.f, \textbf{assoc}) = FIX(f, \textbf{assoc}, \text{FALSE-BDD})$.
- $B(\nu Q.f, \textbf{assoc}) = FIX(f, \textbf{assoc}, \text{TRUE-BDD})$.

The OBDDs for the boolean functions *False* and *True* are denoted by FALSE-BDD and TRUE-BDD respectively. Notice that f has an extra free relational variable Q. *FIX* is described in Figure 7.3. This procedure is similar to *Lfp* and *Gfp*, described in Section 6.1.

We now give a short example to illustrate our point. Let the state space S be encoded by n boolean variables $x_1, \cdots, x_n$. Consider the following formula:

```
function FIX(f, assoc, B_Q)

result-bdd := B_Q;
repeat
        old-bdd := result-bdd;
        result-bdd := B(f, assoc⟨Q ← old-bdd⟩);
until (equal(old-bdd, result-bdd));
return(result-bdd);
end function
```

Figure 7.3
Pseudocode for the function *FIX*.

$$f = \mu Z.((q \wedge Y) \vee \langle a \rangle Z)$$

Notice that the variable Y is free in f. Let $\text{OBDD}_q(\vec{x})$ be the interpretation for q. Similarly, the OBDD corresponding to the transition a is $\text{OBDD}_a(\vec{x}, \vec{x}')$. Assume that we are given an association list **assoc** that pairs the OBDD $B_Y(\vec{x})$ with Y. In the routine *FIX* the OBDD result-bdd is initially set to:

$$N^0(\vec{x}) = \text{FALSE-BDD}$$

Let N^i be the value of result-bdd at the i-th iteration in the loop of the function *FIX*. At the end of the iteration the value of result-bdd is given by:

$$N^{i+1}(\vec{x}) = (\text{OBDD}_q(\vec{x}) \wedge B_Y(\vec{x})) \vee \exists \vec{x}'(\text{OBDD}_a(\vec{x}, \vec{x}') \wedge N^i(\vec{x}'))$$

The iteration stops when $N^i(\vec{x}) = N^{i+1}(\vec{x})$.

7.5 Translating CTL into the μ-Calculus

In this section we give a translation of CTL into the propositional μ-calculus. The algorithm Tr takes as its input a CTL formula and outputs an equivalent μ-calculus formula with only one transition a.

- $Tr(p) = p$.
- $Tr(\neg f) = \neg Tr(f)$.
- $Tr(f \wedge g) = Tr(f) \wedge Tr(g)$.
- $Tr(\mathbf{EX}\ f) = \langle a \rangle Tr(f)$.

- $Tr(\mathbf{E}[f\ \mathbf{U}\ g]) = \mu Y.(Tr(g) \vee (Tr(f) \wedge \langle a \rangle Y))$.
- $Tr(\mathbf{EG}\ f) = \nu Y.(Tr(f) \wedge \langle a \rangle Y)$.

Note that any resulting μ-calculus formula is closed. Thus, we can omit the environment e from the translation. For example, $Tr(\mathbf{EG}(\mathbf{E}[p\ \mathbf{U}\ q]))$ is given by the μ-calculus formula

$$\nu Y.(\mu Z.(q \vee (p \wedge \langle a \rangle Z)) \wedge \langle a \rangle Y).$$

We denote the states satisfying f by $[\![f]\!]_M$. Using the techniques described in Section 6.1, it is easy to prove the following theorem.

THEOREM 6 Let $M = (S, T, L)$ be a Kripke structure. Assume that the transition a in the translation algorithm Tr is the relation T of the Kripke structure. Let f be a CTL formula. Then, for all $s \in S$

$$M, s \models f \Leftrightarrow s \in [\![Tr(f)]\!]_M$$

7.6 Complexity Considerations

An important open question concerns the complexity of μ-calculus model checking. The most efficient algorithms currently known for this problem are exponential in the alternation depth of the formula. We conjecture that there is no polynomial-time algorithm for the μ-calculus model checking problem. It is possible to show that the problem is in NP $\cap$ co-NP [22, 112, 178]. If the problem was NP-complete, then NP would be equal to co-NP, which is believed to be unlikely. This suggests that it would be very difficult to prove our conjecture.

In order to see that the μ-calculus model checking problem is in NP $\cap$ co-NP, consider the following nondeterministic algorithm that guesses the greatest fixpoints and computes the least fixpoints by iteration, starting with the most deeply nested fixpoint. The guess for a greatest fixpoint can be easily checked to see that it is a fixpoint. Furthermore, although we cannot verify that it is the *greatest* fixpoint, we know that the greatest fixpoint must contain any verified guess. By monotonicity, the final value computed by this nondeterministic algorithm will be a subset of the real interpretation of the formula. Moreover, there is a run of the algorithm which calculates the set of states satisfying the μ-calculus formula. Thus, a state s satisfies the formula if and only if it is in the set computed by some run of the algorithm. Consequently, the model checking problem for the μ-calculus formula is in NP. Note that we can negate formulas, so the complexity of determining if a state satisfies a formula is the same as the complexity of determining if a state does not satisfy the formula. Hence, the problem is in the intersection of NP and co-NP.

8 Model Checking in Practice

8.1 The SMV Model Checker

SMV ("*Symbolic Model Verifier*") [191] is a tool for checking that finite-state systems satisfy specifications given in CTL. It uses the OBDD-based symbolic model checking algorithm in Section 6.2. The language component of SMV is used to describe complex finite-state systems. Some of the most important features of the language are described below:

Modules The user can decompose the description of a complex finite-state system into modules. Individual modules can be instantiated multiple times, and modules can reference variables declared in other modules. Standard visibility rules are used for naming variables in hierarchically structured designs. Modules can have parameters, which may be state components, expressions, or other modules. Modules can also contain fairness constraints which can be arbitrary CTL formulas (See Section 6.3).

Synchronous and interleaved composition SMV modules can be composed either synchronously or using interleaving. In a synchronous composition, a single step in the composition corresponds to a single step in each of the components. With interleaving, a step of the composition represents a step by exactly one component. If the keyword `process` precedes an instance of a module, interleaving is used; otherwise synchronous composition is assumed.

Nondeterministic transitions The state transitions in a model may be either deterministic or *nondeterministic*. Nondeterminism can reflect actual choice in the actions of the system being modeled, or it can be used to describe a more abstract model where certain details are hidden. The ability to specify nondeterminism is missing from many hardware description languages, but it is crucial when making high-level models.

Transition relations The transition relations of modules can be specified either explicitly in terms of boolean relations on the current and next state values of state variables, or implicitly as a set of parallel assignment statements. The parallel assignment statements define the values of variables in the next state in terms of their values in the current state.

We will not provide a formal syntax or semantics for the language here; these can be found in McMillan's thesis [191]. Instead, we consider a simple two-process mutual exclusion program (Figure 8.1). Each process can be in one of three code regions: the *noncritical region*, the *trying region*, or the *critical region*. Initially, both processes are in their noncritical regions. The goal of the program is to exclude the possibility that both processes are in their critical regions at the same time. We also require that a process which wants to enter its critical region will eventually be able to do so. A process indicates that it wants to enter its critical region by first entering its trying region. If one process is in

```
MODULE main  --two process mutual exclusion program

VAR
s0: noncritical, trying, critical;
s1: noncritical, trying, critical;
turn: boolean;
pr0: process prc(s0, s1, turn, 0);
pr1: process prc(s1, s0, turn, 1);

ASSIGN
init(turn) := 0;

FAIRNESS   !(s0 = critical)
FAIRNESS   !(s1 = critical)

SPEC   EF((s0 = critical) & (s1 = critical))
SPEC   AG((s0 = trying) -> AF (s0 = critical))
SPEC   AG((s1 = trying) -> AF (s1 = critical))
SPEC   AG((s0 = critical) -> A[(s0 = critical) U
       (!(s0 = critical) & !E[!(s1 = critical) U (s0 = critical)])])
SPEC   AG((s1 = critical) -> A[(s1 = critical) U
       (!(s1 = critical) & !E[!(s0 = critical) U (s1 = critical)])])

MODULE prc(state0, state1, turn, turn0)

ASSIGN
init(state0) := noncritical;
next(state0) :=
  case
   (state0 = noncritical) : trying,noncritical;
   (state0 = trying) & (state1 = noncritical): critical;
   (state0 = trying) & (state1 = trying) & (turn = turn0):  critical;
   (state0 = critical) : critical,noncritical;
   1: state0;
  esac;
next(turn) :=
  case
   turn = turn0 & state0 = critical: !turn;
   1: turn;
  esac;

FAIRNESS   running
```

Figure 8.1
SMV code for two-process mutual exclusion program.

its trying region and the other is in its noncritical region, the first process can immediately enter its critical region. If both processes are in their trying regions, the boolean variable *turn* is used to determine which process enters its critical region. If the value of *turn* is 0, then process 0 can enter its critical region and change the value of *turn* to 1. If the value of *turn* is 1, then process 1 can enter its critical region and change the value to 0. We assume that a process must eventually leave its critical region; however, it may remain in its noncritical region forever.

To describe the syntax of SMV in more detail, consider the program in Figure 8.1. Module definitions begin with the keyword `MODULE`. The module `main` is the top-level module. The module `prc` has formal parameters `state0`, `state1`, `turn`, and `turn0`. Variables are declared using the keyword `VAR`. In the example, `turn` is a boolean variable, while `s0` and `s1` are variables which can have one of the following values: `noncritical`, `trying` or `critical`. The `VAR` statement is also used to instantiate other modules as shown on lines 6 and 7. In our example, the module `prc` is instantiated twice, once with the name `pr0` and once with the name `pr1`. Because the keyword `process` is used in both cases, the global model is constructed by interleaving steps from `pr0` and `pr1`.

The `ASSIGN` statement is used to define the initial states and transitions of the model. In this example, the initial value of the boolean variable `turn` is `0`. The value of the variable `state0` in the next state is given by the `case` statement in lines 23–29. The value of `turn` in the next state is given by the `case` statement in lines 31–34. The value of a `case` statement is determined by evaluating the clauses within the statement in sequence. Each clause consists of a condition and an expression, which are separated by a colon. If the condition in the first clause holds, the value of the corresponding expression determines the value of the `case` statement. Otherwise, the next clause is evaluated. An expression may be a set of values (e.g., 24 and 27). When a set expression is assigned to a variable, the value of the variable is chosen nondeterministically from the set.

Fairness constraints are given by `FAIRNESS` statements. Using the proposition `running` in the fairness constraint for module `prc` restricts the considered computations to only those in which each instance of `prc` is executed infinitely often. Without imposing additional constraints, the nondeterministic choice in line 27 would allow a process to remain in its critical region forever. The fairness constraints in lines 10 and 11 are used to prevent this possibility. The CTL properties to be verified are given as `SPEC` statements. The first specification (line 12) checks for a violation of the mutual exclusion requirement. The second and third specifications (lines 13 and 14) check that a process which wants to enter its critical region will eventually be able to do so. The last two specifications (lines 15 and 17) check whether processes must strictly alternate entry into their critical regions. More specifically, when a process leaves its critical region, there is no path by which it can reenter without the other process first entering the critical section.

```
-- specification EF (s0 = critical & s1 = critical) is false

-- specification AG (s0 = trying -> AF s0 = critical) is true

-- specification AG (s1 = trying -> AF s1 = critical) is true

-- specification AG (s0 = critical -> A(... is false

-- specification AG (s1 = critical -> A(... is false

resources used:
user time: 1.15 s, system time: 0.3 s
BDD nodes allocated: 2405
BDD nodes representing transition relation: 56 + 1
```

Figure 8.2
Output generated by SMV for mutual exclusion program.

When SMV is run on the program in Figure 8.1, the output in Figure 8.2 is produced. Note that the mutual exclusion is not violated and that absence of starvation is guaranteed. Because the last two specifications are false, strict alternation of critical regions is unnecessary. SMV produced counterexample computation paths in the last two cases. One of the counterexamples is included in Figure 8.3. This counterexample demonstrates that process 0 can enter its critical region several times without process 1 entering its critical region. The computation path is described as a sequence of changes to state variables. Thus, if a state variable is not mentioned in a state it means that its value has not been changed. Although the first specification is false, no counterexample is generated. The negation of a formula with an existential path quantifier will have a universal path quantifier. Therefore, no single computation path can serve as a counterexample.

8.2 A Realistic Example

This section briefly describes the formalization and verification of the cache coherence protocol described in the draft IEEE Futurebus+ standard (IEEE Standard 896.1–1991) [147]. A precise model of the protocol was constructed in the SMV language and model checking was used to show that it satisfied a formal specification of cache coherence. In the process of formalizing and verifying the protocol, a number of errors and ambiguities were discovered. This was the first time that formal methods have been used to find nontrivial errors in a proposed IEEE standard. The result of this project is a concise, comprehensible and

```
-- specification AG (s0 = critical -> A(... is false
-- as demonstrated by the following execution sequence
state 2.1: s0 = noncritical
           s1 = noncritical
           turn = 0

state 2.2: [executing process pr0]

state 2.3: [executing process pr0]
           s0 = trying

state 2.4: s0 = critical

state 2.5: [executing process pr0]

state 2.6: s0 = noncritical
           turn = 1

state 2.7: [executing process pr0]

state 2.8: [executing process pr0]
           s0 = trying

state 2.9: s0 = critical
```

Figure 8.3
Counterexample for strict alternation of critical regions.

unambiguous model of the cache coherence protocol. This experience demonstrates that hardware description languages and model checking techniques can be used to help design real industrial standards. For a more detailed treatment of this example, the reader is referred to the paper [66], which deals exclusively with this topic.

Futurebus+ is a bus architecture for high-performance computers. The goal of the committee that developed Futurebus+ was to create a public standard for bus protocols that was unconstrained by the characteristics of any particular processor or device technology and that would be widely accepted and implemented by vendors. The cache coherence protocol used in Futurebus+ is required to insure consistency of data in hierarchical systems composed of many processors and caches interconnected by multiple bus segments. Such protocols are notoriously complex, and therefore quite difficult to debug. Futurebus+ is, in fact, the first bus standard to include this capability. Previous attempts to validate the

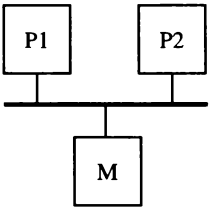


Figure 8.4
Single bus system.

protocol had been based entirely on informal techniques [101]. In particular, no attempt had been made to specify the entire protocol formally or to analyze it using an automatic verification system.

The major part of the project involved developing a formal model for the cache coherence protocol in the SMV language and deriving CTL specifications for its correctness from the textual description of the protocol in the standard. The model for the cache coherence protocol consists of 2300 lines of SMV code (not counting comments). The model is highly nondeterministic, both to reduce the complexity of verification (by hiding details) and to cover allowed design choices (indicated in the standard using the word *may*). By using SMV several potential errors were found in the hierarchical protocol. The largest configuration that was verified had three bus segments, eight processors, and over 10^{30} states.

The Futurebus+ protocol maintains coherence by having the individual caches *snoop*, or observe, all bus transactions. Coherence across buses is maintained using *bus bridges*. Special agents at the ends of the bridges represent remote caches and memories. In order to increase performance, the protocol uses *split transactions*. When a transaction is split, its completion is delayed and the bus is freed; at some later time, an explicit response is issued to complete the transaction. This facility makes it possible to service local requests while remote requests are being processed.

To demonstrate how the protocol works, we consider some example transactions for a single *cache line* in the two-processor system shown in Figure 8.4. A cache line is a series of consecutive memory locations that is treated as a unit for coherence purposes. Initially, neither processor has a copy of the line in its cache; they are said to be in the *invalid* state. Processor P1 issues a *read-shared* transaction to obtain a readable copy of the data from memory M. Processor P2 snoops this transaction, and may, if it wishes, also obtain a readable copy; this is called *snarfing*. If P2 snarfs, then at the end of the transaction, both caches contain a *shared-unmodified* copy of the data. Next, P1 decides to write to a location in the cache line. In order to maintain coherence, the copy held by

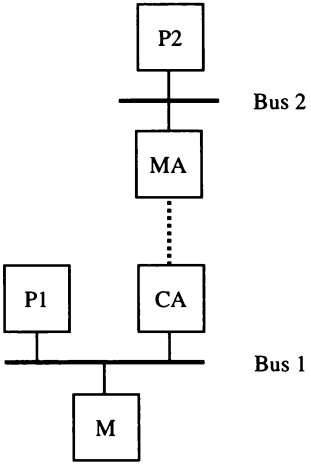


Figure 8.5
Two bus system.

P2 must be eliminated. Processor P1 issues an *invalidate* transaction on the bus. When P2 snoops this transaction, it purges the line from its cache. At the end of the invalidate, P1 now has an *exclusive-modified* copy of the data. The standard specifies the possible states of the cache line within each processor and how this state is updated during each possible transaction.

We now consider a two-bus example to illustrate how the protocol works in hierarchical systems; see Figure 8.5. Initially, both processor caches are in the invalid state. If processor P2 issues a *read-modified* to obtain a writable copy of the data, then the memory agent MA on bus 2 splits the transaction, for it must get the data from the memory on bus 1. The command is passed down to the cache agent CA, and CA issues the read-modified on bus 1. Memory M supplies the data to CA, which in turn passes it to MA. MA now issues a *modified-response* transaction on bus 2 to complete the original split transaction. Suppose now that P1 issues a read-shared command on bus 1. CA, knowing that a remote cache has an exclusive-modified copy, *intervenes* in the transaction to indicate that it will supply the data, and splits the transaction, in that it must obtain the data from the remote cache. CA passes the read-shared to MA, which issues it on bus 2. P2 intervenes and supplies the data to MA, which passes it to CA. The cache agent performs a *shared-response* transaction which completes the original read-shared issued by P1. The standard contains an English description of the hierarchical protocol, but does not specify the interaction between the cache agents and memory agents.

The Protocol Specification [147] contains two sections dealing with the cache coherence protocol. The first, a description section, is written in English and contains an informal and readable overview of how the protocol operates, but it does not cover all scenarios. The second, a specification section, is intended to be the real standard. This section is written using *boolean attributes*. A boolean attribute is essentially a boolean variable together with some rules for setting and clearing it. The attributes are more precise, but they are difficult to read. The behavior of an individual cache or memory is given in terms of roughly three hundred attributes, of which about forty-five deal with cache coherence.

In order to make the verification feasible, it was necessary to use a number of abstractions. First, a number of the low-level details dealing with how modules communicate were eliminated. The most significant simplification was to use a model in which one step corresponds to one transaction on one of the buses in the system. This allowed us to hide all of the handshaking necessary to issue a command. Another example concerns the bus arbitration. The standard specifies two arbitration schemes, but a model in which the bus master is chosen completely nondeterministically was used in the verification. In addition, the standard describes how models behave in various exceptional situations, such as when a parity error is observed on the data bus. However, such conditions were not considered.

The second class of simplifications was used to reduce the size of some parts of the system. For example, only transactions involving a single cache line were considered. This is sufficient since transactions involving one cache line cannot affect the transactions involving a different cache line. Also, the data in each cache line were reduced to a single bit. The third class of simplifications involved eliminating the *read-invalid* and *write-invalid* commands. These commands are used in DMA transfers to and from memory. The protocol does not guarantee coherence for a cache line when a write-invalid transaction is issued for that line.

The last class of abstractions involved using nondeterminism to simplify the models of some of the components. For example, processors are assumed to issue read and write requests for a given cache line nondeterministically. Responses to split transactions are assumed to be issued after arbitrary delays. Finally, the model of a bus bridge is highly nondeterministic.

Figure 8.6 shows a part of the SMV program used to model the processor caches. This code determines how the state of the cache line is updated. Within this code, state components with upper-case names (`CMD`, `SR`, `TF`) denote bus signals visible to the cache, and components with lower-case names (`state`, `tf`) are under the control of the cache. The first part of the code (lines 3–13) specifies what may happen when an idle cycle occurs (`CMD=none`). If the cache has a shared-unmodified copy of the line, then the line may be nondeterministically kicked out of the cache unless there is an outstanding request to

```
next(state) :=
  case
  CMD=none:
    case
    state=shared-unmodified:
      case
      requester=exclusive: shared-unmodified;
      1: invalid, shared-unmodified;
      esac;
    state=exclusive-unmodified: invalid, shared-unmodified,
      exclusive-unmodified, exclusive-modified;
    1: state;
    esac;
  .
  .
  .
  master:
    case
    CMD=read-shared:  -- Cache issues a read-shared
      case
      state=invalid:
        case
        !SR & !TF: exclusive-unmodified;
        !SR: shared-unmodified;
        1: invalid;
        esac;
      .
      .
      .
      esac;
    .
    .
    .
    esac;
  .
  .
  .
  CMD=read-shared:  -- Cache observes a read-shared
    case
    state in invalid, shared-unmodified:
      case
      !tf: invalid;
      !SR: shared-unmodified;
      1: state;
      esac;
    .
    .
    .
    esac;
  .
  .
  .
  esac;
```

Figure 8.6
A portion of the processor cache model.

change the line to exclusive-modified. If a cache has an exclusive-unmodified copy of the line, it may kick the line out of the cache or change it to exclusive-modified.

The second part of the code (lines 15–26) indicates how the cache line state is updated when the cache issues a read-shared transaction (`master` and `CMD=read-shared`). This should only happen when the cache does not have a copy of the line. If the transaction is not split (`!SR`), then the data will be supplied to the cache. Either no other caches will snarf the data (`!TF`), in which case the cache obtains an exclusive-unmodified copy, or some other cache snarfs the data, and everyone obtains shared-unmodified copies. If the transaction is split, the cache line remains in the invalid state.

The last piece of code (lines 30–39) tells how caches respond when they observe another cache issuing a read-shared transaction. If the observing cache is either invalid or has a shared-unmodified copy, then it may indicate that it does not want a copy of the line by deasserting its `tf` output. In this case, the line becomes invalid. Alternatively, the cache may assert `tf` and try to snarf the data. In this case, if the transaction is not split (`!SR`), the cache obtains a shared-unmodified copy. Otherwise, the cache stays in its current state.

Next, we discuss the specifications used in verifying the protocol. More exhaustive specifications are obviously possible; in particular, we have only tried to describe what cache coherence is, not how it is achieved. The first class of properties states that if a cache has an exclusive-modified copy of some cache line, then all other caches should not have copies of that line. The specification includes the formula

$$\mathbf{AG}(p1.writable \rightarrow \neg p2.readable)$$

for each pair of caches *p1* and *p2*. Here, *p1.writable* is a macro expression (given by a `DEFINE` statement in SMV), which is true when *p1* is in the exclusive-modified state. Similarly, *p2.readable* is true when *p2* is not in the invalid state.

Consistency is described by requiring that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}(p1.readable \wedge p2.readable \rightarrow p1.data = p2.data)$$

Similarly, if memory has an up-to-date copy of the line, then any cache that has a copy must agree with memory on the data.

$$\mathbf{AG}(p.readable \wedge \neg m.memory - line - modified \rightarrow p.data = m.data)$$

The variable *m.memory-line-modified* is false when memory has an up-to-date copy of the cache line.

The final class of properties is used to check that it is always possible for a cache to get read or write access to the line.

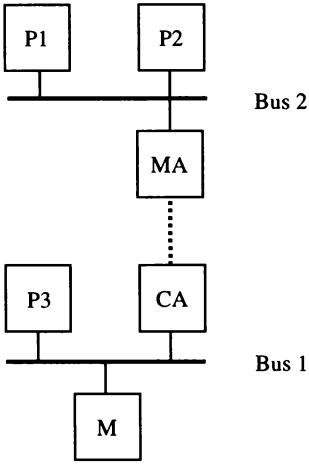


Figure 8.7
Error in hierarchical configuration.

AG EF *p.readable* $\wedge$ **AG EF** *p.writable*

Finally, we describe two of the errors that were found while trying to verify the protocol. The first error occurs in the single bus protocol. Consider the system shown in Figure 8.4. The following scenario is not excluded by the standard. Initially, both caches are invalid. Processor P1 obtains an exclusive-unmodified copy. Next, P2 issues a read-modified, which P1 splits for invalidation. The memory M supplies a copy of the cache line to P2, which transitions to the shared-unmodified state. At this point, P1, still having an exclusive-unmodified copy, transitions to exclusive-modified and writes the cache line. P1 and P2 are now inconsistent. This bug can be fixed by requiring that P1 transition to the shared-unmodified state when it splits the read-modified for invalidation. The change also fixes a number of related errors.

The second error occurs in the hierarchical configuration shown in Figure 8.7. P1, P2, and P3 all obtain shared-unmodified copies of the cache line. P1 issues an invalidate transaction that P2 and MA split. P3 issues an invalidate that CA splits. The bus bridge detects that an *invalidate-invalidate collision* has occurred. That is, P3 is trying to invalidate P1, while P1 is trying to invalidate P3. When this happens, the standard specifies that the collision should be resolved by having the memory agent invalidate P1. When the memory agent tries to issue an invalidate for this purpose, P2 sees that there is already a transaction in progress for this cache line and asserts a busy signal on the bus. MA observes this and acquires the *requester-waiting* attribute. When a module has this attribute, it will wait until

it sees a completed response transaction before retrying its command. P2 now finishes invalidating and issues a modified-response. This is split by MA because P3 is still not invalid. However, MA still maintains the requester-waiting attribute. At this point, MA will not issue commands since it is waiting for a completed response, but no such response can occur. The deadlock can be avoided by having MA clear the requester-waiting attribute when it observes that P2 has finished invalidating.

9 Model Checking and Automata Theory

In this chapter we present some basic facts from automata theory and demonstrate how model checking can be performed in this framework. In particular, we show how to translate an LTL formula into an automaton. This gives an alternative model checking algorithm for LTL, which can be performed on the fly. In this approach the checked property guides the construction of the state graph for the modeled system. Consequently, it may be possible to avoid constructing large parts of the state graph.

9.1 Automata on Finite and Infinite Words

A finite automaton is a mathematical model of a device that has a constant amount of memory, independent of the size of its input. We will consider finite automata over finite words and finite automata over infinite words (also called *ω-automata*).

Formally, a finite automaton (over finite words) $\mathcal{A}$ is a five tuple $\langle \Sigma, Q, \Delta, Q^0, F \rangle$ such that

- Σ is the finite *alphabet*.
- Q is the finite set of *states*.
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*.
- $Q^0 \subseteq Q$ is the set of *initial states*.
- $F \subseteq Q$ is the set of *final states*.

An automaton can be represented as a graph with labeled transitions, in which the set of nodes is Q and the edges are given by Δ. An example of an automaton is shown in Figure 9.1. There, $\Sigma = \{a, b\}$, $Q = \{q_1, q_2\}$, $Q^0 = \{q_1\}$ (initial states are marked with an incoming arrow), and $F = \{q_1\}$ (accepting states are marked with a double circle).

Let v be a word (string, sequence) of Σ^* of length $|v|$. A *run* of $\mathcal{A}$ over v is a mapping $\rho : \{0, 1, \ldots, |v|\} \mapsto Q$ such that:

- The first state is an initial state, that is, $\rho(0) \in Q^0$.
- Moving from the ith state $\rho(i)$ to the $i + 1$st state $\rho(i + 1)$ upon reading the ith input letter $v(i)$ is consistent with the transition relation. That is, for $0 \leq i < |v|$ $(\rho(i), v(i), \rho(i + 1)) \in \Delta$.

A run ρ of $\mathcal{A}$ on v corresponds to a path in the automaton graph from an initial state $\rho(0)$ to a state $\rho(|v|)$, where the edges on this path are labeled according to the letters in v. We say that v is an *input* to the automaton $\mathcal{A}$ or that $\mathcal{A}$ *reads* v. A run ρ over v is *accepting* if it ends in an accepting state, that is, $\rho(|v|) \in F$. An automaton $\mathcal{A}$ *accepts* a

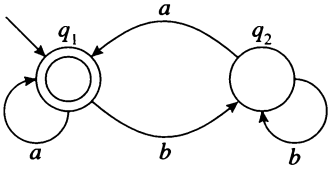


Figure 9.1
A finite automaton.

word v if and only if there exists an accepting run of $\mathcal{A}$ on v. For example, the automaton in Figure 9.1 accepts the word *aabba* because there is a run that passes through the states $q_1q_1q_1q_2q_2q_1$.

The *language* of $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ consists of all the words accepted by $\mathcal{A}$. The automaton in Figure 9.1 accepts the language described by the regular expression $\varepsilon + (a + b)^*a$, that is, either the empty word ε, or words that consist of any number of a's or b's and end with an a. The operator $+$ indicates a choice, and the $*$ operator indicates any finite number of repetitions.

Because most concurrent systems are designed not to halt during normal execution, we model computations as infinite sequences of states. Thus, this chapter will focus on finite automata over infinite words. These automata have the same structure as finite automata over finite words. However, they recognize words from Σ^ω, where the superscript ω indicates an infinite number of repetitions.

The simplest automata over infinite words are Büchi [39] automata. A Büchi automaton has the same components as an automaton over finite words. However, F is called the set of *accepting states*, rather than final states. A run of a Büchi automaton $\mathcal{A}$ over an infinite word $v \in \Sigma^\omega$ is defined in almost the same way as a run of a finite automaton over a finite word, except that now $|v| = \omega$. Thus, the domain of a run is the set of all natural numbers. Again, a run corresponds to a path in the graph of the automaton, but the path is now an infinite one.

Let *inf* (ρ) be the set of states that appear infinitely often in the run ρ (when treating the run as an infinite path). A run ρ of a Büchi automaton $\mathcal{A}$ over an infinite word is *accepting* if and only if *inf* $(\rho) \cap F \neq \emptyset$, that is, when some accepting state appears in ρ infinitely often.

The structure shown in Figure 9.1 can be interpreted as a Büchi automaton. In this case one of the words it accepts is $(ab)^\omega$, that is, an infinite sequence of alternating a's and b's, starting with an a. The language it accepts is the set of words with *infinitely* many a's, which can be written as the ω-regular expression $(b^*a)^\omega$.

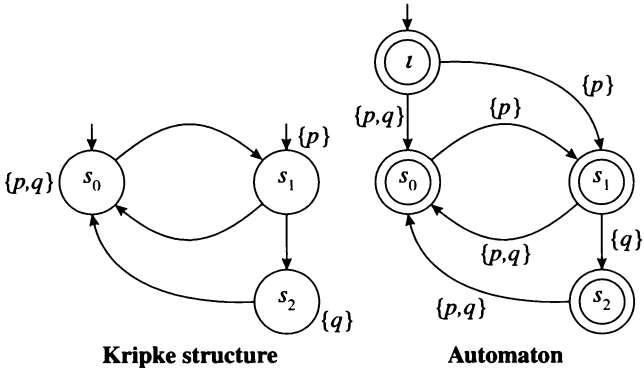


Figure 9.2
Transforming a Kripke structure into an automaton.

9.2 Model Checking Using Automata

Finite automata can be used to model concurrent and interactive systems. Either the state Q or the alphabet Σ can then represent the states of the modeled system. One of the main advantages of using automata for model checking is that both the modeled system and the specification are represented in the same way. A Kripke structure directly corresponds to an ω-regular automaton, where all the states are accepting. Then, the set of behaviors of a system M is the language $\mathcal{L}(\mathcal{A})$ of the corresponding automaton $\mathcal{A}$. Specifically, a Kripke structure $\langle S,\ R,\ S_0,\ L\rangle$ where $L : S \rightarrow 2^{AP}$, can be transformed into an automaton $\mathcal{A} = \langle \Sigma,\ S \cup \{\iota\},\ \Delta,\ \{\iota\},\ S \cup \{\iota\}\rangle$, where $\Sigma = 2^{AP}$. We have $(s, \alpha, s') \in \Delta$ for $s,\ s' \in S$ if and only if $(s, s') \in R$ and $\alpha = L(s')$. In addition, $(\iota, \alpha,\ s) \in \Delta$ if and only if $s \in S_0$ and $\alpha = L(s)$. Figure 9.2 shows a Kripke structure and its corresponding automaton.

The specification can also be given as an automaton $\mathcal{S}$, over the same alphabet. Then, $\mathcal{L}(\mathcal{S})$ is the set of allowed behaviors. We will present several examples of properties expressed using Büchi automata. The properties refer to the mutual exclusion example in Figure 2.2. In these examples, we annotate edges with boolean expressions rather than a subset of the propositions AP. Each edge may represent several transitions, where each transition corresponds to a truth assignment for AP that satisfies the boolean expression. For example, when $AP = \{X, Y, Z\}$, an edge labeled $X \wedge \neg Y$ matches the transitions labeled with $\{X, Z\}$ and $\{X\}$ (that is, the sets of propositions that include X and do not include Y but may or may not include Z).

The set of atomic propositions AP in the following examples corresponds to the labels CR_0 and CR_1 of the mutual exclusion example. For instance, the proposition CR_0 holds in

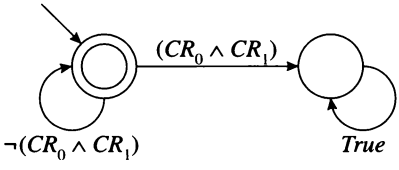


Figure 9.3
Mutual exclusion property.

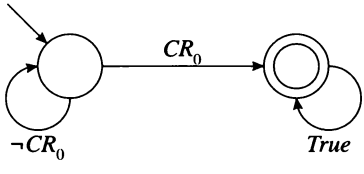


Figure 9.4
A liveness property.

the states where the program counter of process P_0 is CR_0. Figure 9.3 shows an automaton that specifies the property that the two processes cannot enter their critical section at the same time. This specification is given by the LTL path formula $\mathbf{G}\ \neg(CR_0 \wedge CR_1)$. The property obviously holds for the mutual exclusion example.

Figure 9.4 shows an automaton that specifies the property that the process P_0 will eventually enter its critical section, and is given by the LTL path formula $\mathbf{F}\ CR_0$. This property does not hold in our example system, for it is possible that P_0 never attempts to enter its critical section.

The system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S}) \tag{9.1}$$

That is, each behavior of the modeled system is among the behaviors that are allowed by the specification. Let $\overline{\mathcal{L}(\mathcal{S})}$ be the language $\Sigma^\omega - \mathcal{L}(\mathcal{S})$. Then (9.1) can be rewritten as

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset \tag{9.2}$$

This means that there is no behavior of $\mathcal{A}$ that is disallowed by $\mathcal{S}$. If the intersection is not empty, any behavior in it corresponds to a counterexample.

Büchi automata are closed under intersection and complementation [39]. This means that there exists an automaton that accepts exactly the intersection of the languages of two automata, and an automaton that recognizes exactly the complement of the language of

a given automaton. We will later show how to construct an automaton that recognizes the intersection of two languages accepted by a pair of Büchi automata. The details of computing the complement of a Büchi automaton are rather involved. Constructions for this purpose can be found in [226, 234].

The formulation of the correctness criterion in (9.2) suggests the following model-checking procedure:

1. Complement the automaton $\mathcal{S}$, that is, construct an automaton $\overline{\mathcal{S}}$ that recognizes the language $\overline{\mathcal{L}(\mathcal{S})}$.
2. Construct the automaton that accepts the intersection of the languages $\mathcal{L}(\mathcal{A})$ and $\overline{\mathcal{L}(\mathcal{S})}$.

If the intersection is empty, announce that the specification $\mathcal{S}$ holds for $\mathcal{A}$. Otherwise, we must provide a counterexample. We will show later that an infinite word in the intersection can be represented in a finitary way. Specifically, there is a counterexample of the form $u\ v^{\omega}$ where u and v are finite words.

In some implementations such as SPIN [138, 140], the user is supposed to provide the automaton for the complement of $\mathcal{S}$ directly instead of providing the automaton for $\mathcal{S}$. In this approach, the user specifies the bad behaviors rather than the good ones. Another possibility [162] is to use a different type of ω-regular automata, for which complementation is easy.

Finally, the automaton $\mathcal{S}$ may be obtained using a translation from some specification language such as LTL. In this case, instead of translating a property φ into $\mathcal{S}$ and then complementing $\mathcal{S}$, we can simply translate $\neg\varphi$, which immediately provides an automaton for the complement language, as required in (9.2). Later, we will provide an efficient translation from LTL to Büchi automata.

Let $\mathcal{B}_1 = \langle \Sigma,\ Q_1,\ \Delta_1,\ Q_1^0,\ F_1 \rangle$ and $\mathcal{B}_2 = \langle \Sigma,\ Q_2,\ \Delta_2,\ Q_2^0,\ F_2 \rangle$. We can build an automaton that accepts $\mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$ as follows: $\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma,\ Q_1 \times Q_2 \times \{0, 1, 2\},\ \Delta,\ Q_1^0 \times Q_2^0 \times \{0\},\ Q_1 \times Q_2 \times \{2\} \rangle$. We have $(\langle r_i, q_j, x \rangle, a, \langle r_m, q_n, y \rangle) \in \Delta$ if and only if the following conditions hold:

- $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$, that is, the local components agree with the transitions of $\mathcal{B}_1$ and $\mathcal{B}_2$.
- The third component is affected by the accepting conditions of $\mathcal{B}_1$ and $\mathcal{B}_2$.
  - if $x = 0$ and $r_m \in F_1$, then $y = 1$.
  - if $x = 1$ and $q_n \in F_2$, then $y = 2$.
  - if $x = 2$ then $y = 0$.
  - otherwise, $y = x$.

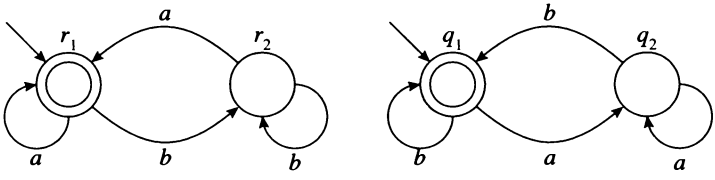


Figure 9.5
An automaton for infinite number of a's (left) and an automaton for an infinite number of b's (right).

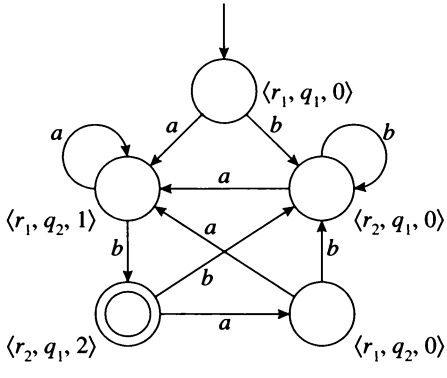


Figure 9.6
An automaton for words with an infinite number of a's and b's.

The third component is responsible for guaranteeing that accepting states from both $\mathcal{B}_1$ and $\mathcal{B}_2$ appear infinitely often. Note that accepting states from both automata may appear together only finitely many times even if they appear individually infinitely often. Hence setting $F = F_1 \times F_2$ does not work. The third component is initially 0. It changes from 0 to 1 when an accepting state of the first automaton is seen. It changes from 1 to 2 when an accepting state of the second automaton is seen, and in the next state, returns back to 0. The constructed automaton accepts exactly when infinitely many states from F_1 and infinitely many states from F_2 occur. The intersection of the automata in Figure 9.5 appears in Figure 9.6. Only nodes reachable from the initial state are shown.

A simpler intersection is obtained when all of the states of one of the automata are accepting. Such an intersection is used, for instance, in Equation 9.2, because all the states of the automaton for the modeled system are accepting. Assume all of the states of $\mathcal{B}_1$ are accepting and that the acceptance set of $\mathcal{B}_2$ is F_2. Their intersection will be defined as follows:

$$\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma,\ Q_1 \times Q_2,\ \Delta',\ Q_1^0 \times Q_2^0,\ Q_1 \times F_2 \rangle$$

The accepting states are pairs from $Q_1 \times F_2$ in which the second component is an accepting state. Moreover, $(\langle r_i, q_j \rangle, a, \langle r_m, q_n \rangle) \in \Delta'$ if and only if $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$.

The general algorithm for computing intersection is useful for verifying systems with fairness constraints. In this case, some of the states of the system automaton $\mathcal{B}_1$ may not be accepting.

9.2.1 Nondeterministic Büchi Automata

For both regular and Büchi automata, we allow the transition relation Δ to be nondeterministic. That is, there can be transitions (q, a, l), $(q, a, l') \in \Delta$, where $l \neq l'$. Any nondeterministic finite automaton on *finite words* can be translated into an equivalent deterministic automaton, that is, one that accepts the same language. This is done using the *subset construction*. For a nondeterministic automaton $\mathcal{M} = \langle \Sigma,\ Q,\ \Delta,\ Q^0,\ F \rangle$, we construct an equivalent deterministic automaton $\mathcal{M}' = \langle \Sigma,\ 2^Q,\ \Delta',\ \{Q^0\},\ F' \rangle$, such that $\Delta' \subseteq 2^Q \times \Sigma \times 2^Q$ contains (Q_1, a, Q_2) where

$$Q_2 = \bigcup_{q \in Q_1} \{q' \mid (q, a, q') \in \Delta\}.$$

The set F' is defined as $\{Q' \mid Q' \subseteq Q \wedge Q' \cap F \neq \emptyset\}$. Because $\mathcal{M}'$ is deterministic, Δ' can be represented as a function $\Delta' : 2^Q \times \Sigma \to 2^Q$. Each state of $\mathcal{M}'$ corresponds to the set of states that $\mathcal{M}$ can reach after reading some given input sequence.

Complementing a nondeterministic automaton over finite words can be performed by first determinizing it using the subset construction. Then, we interchange the accepting and the nonaccepting states. However, for Büchi automata the situation is different. Not every Büchi automaton has an equivalent deterministic Büchi automaton. A language recognized by a deterministic Büchi automaton $\mathcal{B}$ satisfies the following condition for each word $v \in \Sigma^\omega$: If there are infinitely many finite prefixes of v whose finite runs reach an accepting state, then v is in the language. If the automaton is deterministic then there is a unique run for each finite prefix of a word. Suppose there are infinitely many finite prefixes of v whose finite runs reach accepting states. Then, these runs are prefixes of the unique run of the automaton on v. By definition, this run must be accepting.

Consider the automaton in Figure 9.7. It accepts the language of infinite words over $\Sigma = \{a, b\}$ that have only finitely many a's. This is a nondeterministic automaton, but there is no deterministic automaton that can recognize this language. If there were a deterministic Büchi automaton that could recognize this language, it would have to reach some accepting

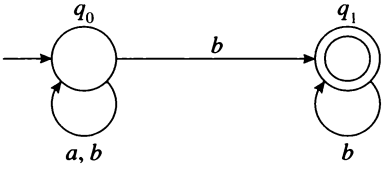


Figure 9.7
An automaton for words with finitely many a's.

state after a finite string b^{n_1} for some $n_1 \geq 0$. Otherwise, the word b^ω could not be accepted. Continuing from this state, this automaton must reach an accepting state after $b^{n_1}ab^{n_2}$, for some $n_2 \geq 0$ (for otherwise $b^{n_1}ab^\omega$ is not accepted), and so forth. Thus, it must accept a word of the form $b^{n_1}ab^{n_2}ab^{n_3}\ldots$, which contains infinitely many a's. It is interesting to note that the complement of this language, that is, the language of infinite words with infinitely many a's can be recognized by a deterministic Büchi automaton (see the automaton on the left in Figure 9.5). Thus, the set of languages accepted by deterministic Büchi automata is not closed under complementation.

9.2.2 Generalized Büchi Automata

Sometimes it is convenient to work with a Büchi automaton with several accepting sets, although this does not extend the set of languages that can be expressed. In particular, we will subsequently describe a translation from an LTL specification into generalized Büchi automaton. A *generalized Büchi automaton* has an acceptance component of the form $F \subseteq 2^Q$. A run ρ of a generalized Büchi automaton is accepting if for each $P_i \in F$, $inf\ (\rho) \cap P_i \neq \phi$. Note that the use of multiple fairness constraints with Kripke structures in Section 4.1.1 corresponds to the notion of acceptance used in generalized Büchi automata.

There is a simple translation from a generalized Büchi automaton $\mathcal{B} = \langle \Sigma, Q, \Delta, Q^0, F\rangle$ to a Büchi automaton. Let $F = \{P_1, \ldots, P_n\}$. Construct

$$\mathcal{B}' = \langle \Sigma,\ Q \times \{0, \ldots, n\},\ \Delta',\ Q^0 \times \{0\},\ Q \times \{n\}\rangle.$$

The transition relation Δ' is constructed such that $(\langle q, x\rangle, a, \langle q', y\rangle) \in \Delta'$ when $(q, a, q') \in \Delta$ and x and y are defined according to the following rules:

- If $q' \in P_i$ and $x = i - 1$ then $y = i$.
- If $x = n$ then $y = 0$.
- Otherwise $x = y$.

The translation expands the size of the automaton by a factor of $n + 1$. Note that in case that the set F of the generalized Büchi automaton is empty, all infinite words over Σ are accepted.

9.3 Checking Emptiness

Checking for emptiness of a Büchi automaton is also simple. Let ρ be an accepting run of a Büchi automaton $\mathcal{B} = \langle \Sigma,\ Q,\ \Delta,\ Q^0,\ F \rangle$. Then, ρ contains infinitely many accepting states from F. Since Q is finite, there is some suffix ρ' of ρ such that every state on it appears infinitely many times. Each state on ρ' is reachable from any other state on ρ'. Hence, the states in ρ' are included in a strongly connected component. This component is reachable from an initial state and contains an accepting state. Conversely, any strongly connected component that is reachable from an initial state and contains an accepting state generates an accepting run of the automaton.

Thus, checking nonemptiness of $\mathcal{L}(\mathcal{B})$ is equivalent to finding a strongly connected component that is reachable from an initial state and contains an accepting state. That is, the language $\mathcal{L}(\mathcal{B})$ is nonempty if and only if there is a reachable accepting state with a cycle back to itself. Clearly, the nodes in such a cycle must belong to some strongly connected component. Conversely, given a strongly connected component with an accepting state, it is always possible to find a cycle through the accepting state. The significance of this observation is that if the language $\mathcal{L}(\mathcal{B})$ is nonempty, then there is a counterexample, which can be represented in a finitary manner. The counterexample is a run, constructed from a finite prefix and a periodic sequence of states. In particular, this applies to the case where $\mathcal{B}$ is the intersection of an automaton that represents the checked system and an automaton that represents the complement of the specification, as described in Section 9.2.

Tarjan's *depth first search* (DFS) algorithm [239] for finding strongly connected components can be used for deciding emptiness of Büchi automata in time $O(|Q| + |\Delta|)$. We will describe an alternative algorithm [84, 141] that is usually more efficient in practice for solving this problem. The algorithm uses *double* DFS for finding cycles with an accepting state.

The two depth first searches are interleaved. The first one can activate the second, and the second search may terminate the entire algorithm or resume the first search from where it has last stopped.

When the first DFS is ready to backtrack from an accepting state after completing the search of its successors, the second search is started, looking for a cycle through this state. If the second search fails to find a cycle, the first search resumes from the point where it was interrupted.

```
procedure emptiness
    for all q0 ∈ Q^0 do
        dfs1(q0);
    terminate(False);
end procedure

procedure dfs1(q)
    local q';
    hash(q);
    for all successors q' of q do
        if q' not in the hash table then dfs1(q');
    if accept(q) then dfs2(q);
end procedure

procedure dfs2(q)
    local q';
    flag(q);
    for all successors q' of q do
        if q' on dfs1 stack then terminate(True);
        else if q' not flagged then dfs2(q');
        end if;
end procedure
```

Figure 9.8
The double DFS algorithm.

In the algorithm in Figure 9.8, we store the state in a hash table when it is discovered by the first DFS and say that the node is *hashed*. A boolean flag is used to indicate whether a state has been encountered by some invocation of the second DFS. If this is the case, then we say that the node is *flagged*. For an efficient implementation, each state in the hash table includes two bits that indicate whether the state is on the search stack of the first or the second DFS.

The algorithm uses the command **terminate** to stop the execution of the entire program and return a value.

When the algorithm terminates with *True*, a cycle through a reachable accepting state is reported as a counterexample for emptiness. Let q_1 be the accepting state with which the second DFS is started. Then the first DFS stack contains a path from an initial state to q_1. This path is the finite prefix of the counterexample. Let q_2 be the state that terminates the

second DFS. The periodic part is constructed as follows: the second DFS stack contains a path from q_1 to q_2; q_2 appears on the search stack of the first DFS, and the states that were inserted on the first DFS stack after q_2 was inserted complete a cycle back to q_1.

Correctness of the Algorithm

The following well known property of DFS is essential for proving the correctness of the algorithm.

LEMMA 23 Let q be a node that does not appear on any cycle. Then the DFS algorithm will backtrack from q only after all the nodes that are reachable from q have been explored and backtracked from.

It is easy to see that this lemma still holds for the first DFS in the double DFS algorithm.

THEOREM 7 The double DFS algorithm returns a counterexample for the emptiness of the checked automaton $\mathcal{B}$ exactly when the language $\mathcal{L}(\mathcal{B})$ is not empty.

Proof When the double DFS returns a path to an accepting state and a cycle through that state, it has found a counterexample for emptiness of the checked automaton. The difficult case is showing that when the algorithm reports emptiness of $\mathcal{L}(\mathcal{B})$, this is indeed the case.

Note that the second DFS flags the states it has reached when started from previous states by the first DFS. Suppose a second DFS is started from a state q and there is a path from q to some state p on the search stack of the first DFS. Then the path from q to p can be completed to a cycle through q, by including the states that appear after p on that stack. There are two cases:

- There exists a path from q to a state on the search stack of the first DFS that contains only unflagged nodes when the second DFS is started from q. In this case, the second DFS will find a cycle as expected.
- On every path from q to a state on the search stack of the first DFS there exists a state r that is already flagged. In this case, the algorithm would not discover a cycle through q.

We will show that the second case is impossible. Suppose the contrary. Then there is an accepting state from which a second DFS starts but fails to find a cycle even though one exists. Let q be the first such state. Let r be the first flagged state that is reached from q during the second DFS and is on a cycle through q. Finally, let q' be the accepting state that starts the second DFS in which r was first encountered. Thus, according to our assumptions, a second DFS was started from q' before a second DFS was started from q. There are two cases (see Figure 9.9):

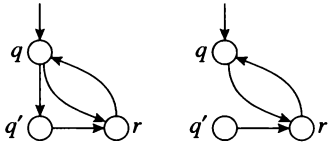


Figure 9.9
The two cases of Theorem 7.

1. *The state q' is reachable from q.* Then there is a cycle $q' \rightarrow \ldots \rightarrow r \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow q'$. This cycle could not have been found previously. Otherwise, the algorithm would already have terminated. However, this contradicts our assumption that q is the first accepting state from which the second DFS missed a cycle.

2. *The state q' is not reachable from q.* If q' appears on a cycle, then a cycle was missed before starting the second DFS from q, contrary to our assumption. According to our assumption, q is reachable from r. Hence, q is reachable from q'. Thus, if q' does not occur on a cycle, by lemma 23 we must have discovered and backtracked from q in the first DFS before backtracking from q'. Hence, according to the double DFS algorithm, we must have started a second DFS from q before starting it from q'. This contradicts our assumption about the order of doing the second DFS. □

9.4 Translating LTL into Automata

In this section we present an algorithm of Gerth, Peled, Vardi, and Wolper [124] for translating an LTL path formula into a generalized Büchi automaton. The algorithm can be thought of as an implementation of the tableau construction for LTL.

In order to apply the following translation procedure we must first put the formula φ into *negation normal form*, in which negation is only applied to propositional variables. First, we rewrite subformulas of the form $\mathbf{F}\,\psi$ as *True* $\mathbf{U}\,\psi$ and subformulas of the form $\mathbf{G}\,\psi$ as *False* $\mathbf{R}\,\psi$. We also use boolean equivalences so that the only boolean operators that are used are *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Finally, negations are pushed inwards, using the LTL equivalences $\neg(\mu\ \mathbf{U}\ \eta) = (\neg\mu)\ \mathbf{R}\ (\neg\eta)$, $\neg(\mu\ \mathbf{R}\ \eta) = (\neg\mu)\ \mathbf{U}\ (\neg\eta)$ and $\neg\,\mathbf{X}\,\mu = \mathbf{X}\,\neg\mu$. For example, consider the formula $(A\ \mathbf{U}\ B) \rightarrow \mathbf{F}\,C$. We replace the implication by a disjunction, obtaining $\neg(A\ \mathbf{U}\ B) \vee \mathbf{F}\,C$. The eventuality $\mathbf{F}\,C$ is replaced by *True* $\mathbf{U}\,C$, obtaining $\neg(A\ \mathbf{U}\ B) \vee (\mathit{True}\ \mathbf{U}\ C)$. Finally, we push the negation inward, resulting in $((\neg A)\ \mathbf{R}\ (\neg B)) \vee (\mathit{True}\ \mathbf{U}\ C)$. In the remainder of the section we will assume that φ is already in normal form.

The basic data structure used in the algorithm is called a *node*. The states of the resulting automaton are represented by nodes. A node q contains the following fields:

record node = [*ID* : NodeID, *Incoming* : NodeID list,
Old : Formula list, *New* : Formula list, *Next* : Formula list];

ID is the unique identifier for the node.

Incoming is the list of predecessor nodes. Each node r on this list represents an edge from r to q.

Old, *New*, *Next* Each of these fields is a list of subformulas of φ. Intuitively, these fields describe temporal properties of suffixes of computations. The subformulas in these fields contain information about a suffix ξ^i of a computation ξ when the following condition holds: ξ^i satisfies all of the subformulas in either *Old* or *New*, and ξ^{i+1} satisfies all of the subformulas in *Next*. The subformulas in *Old* have been already processed by the algorithm at the current node, while the subformulas in *New* have yet to be processed.

We use the symbol $\Leftarrow$ to assign a value to a field of a node. For example, $New \Leftarrow \{\varphi\}$ assigns a singleton set containing the formula φ to the *New* field of the current node.

We maintain the list *Nodes* of nodes whose construction has been completed. These nodes, together with a special node named *init*, constitute the states in the constructed automaton. The node *init* will later become the initial state of the automaton. The list *Nodes* is initially empty.

The algorithm described below uses the function *new_ID*() to generate a new value of type NodeID each time that it is called. The function *Neg* is defined as follows: $Neg(A) = \neg A$, $Neg(\neg A) = A$ for A a proposition, $Neg(True) = False$ and $Neg(False) = True$.

The algorithm for translating the formula φ starts with a single node. This node has a single incoming edge from the special node *init*. In addition, it has $New = \{\varphi\}$ and $Old = Next = \emptyset$.

function *create_graph* (φ)
  expand([*ID* $\Leftarrow$ *new_ID*(),
    Incoming $\Leftarrow$ {*init*},
    Old $\Leftarrow$ $\emptyset$,
    New $\Leftarrow$ $\{\varphi\}$,
    Next $\Leftarrow$ $\emptyset$], $\emptyset$);
end function

For example, the upper node in Figure 9.10 is the one with which the algorithm starts for constructing the automaton for $A\ \mathbf{U}\ (B\ \mathbf{U}\ C)$.

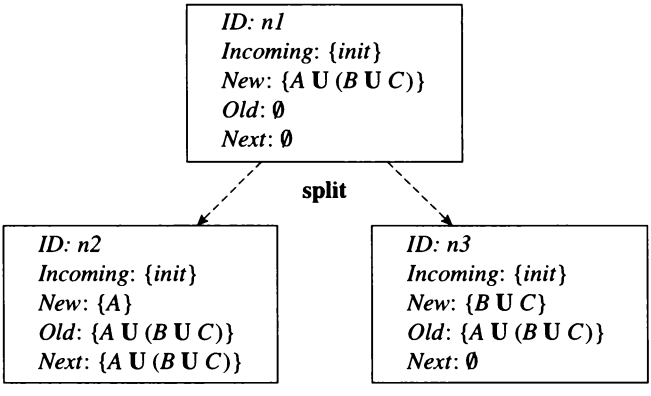


Figure 9.10
Splitting a node.

The recursive function *expand* builds a tableau. It accepts two parameters: the current node and the list of previously constructed nodes and returns a list of nodes.

function *expand* (q, *Nodes*)

For the current node q, the algorithm checks whether the field *New* of q is empty. If so, it checks if the current node can be added to *Nodes*. If there is a node r in *Nodes* with the same subformulas as q in both its *Old* and *Next* fields, the *Incoming* field of r is changed as follows: The list of incoming edges of q are added to the incoming edges of r. If no such node exists in *Nodes*, then q is added to *Nodes*, and a new current node q' is formed as follows:

- There is an edge from q to q', that is, *Incoming* of q' is set to $\{q\}$.
- The field *New* of q' is set to *Next*(q).
- The fields *Old* and *Next* of q' are set to be empty.

Figure 9.11 shows a node q that is put into the list *Nodes*. Then, the new node q' is formed.

if *New*(q) is empty **then**
    if there exists a node r in *Nodes* with
        Old(r) = *Old*(q) **and** *Next*(r) = *Next*(q) **then**
        Incoming(r) := *Incoming*(r) ∪ *Incoming*(q);
        return(*Nodes*);
    else *expand*([*ID* ⇐ *new_ID*(),

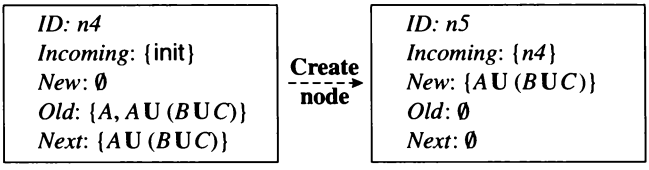


Figure 9.11
Creating a new node.

$$\begin{aligned} &Incoming \Leftarrow \{ID(q)\}, \\ &Old \Leftarrow \emptyset, \\ &New \Leftarrow Next(q), \\ &Next \Leftarrow \emptyset], Nodes \cup \{q\}); \end{aligned}$$

end if;

Otherwise, if the field *New* of q is not empty, a formula η in *New* is selected and removed from *New*. If η already belongs to *Old*, then we simply expand the node without η in *New*.

else /* $New(q)$ is not empty */
let $\eta \in New(q)$;
$New(q) := New(q) - \{\eta\}$;
if $\eta \in Old(q)$ **then**
$expand(q, Nodes)$;

Suppose now that η is not in $Old(q)$. Then according to the main operator of η, the node q is *split* into two parts q_1 and q_2 or is *replaced* by a new version q'. The new nodes are formed by first selecting new names for these nodes and copying from q the fields *Incoming*, *Old*, *New* and *Next*. Then, η is added to the list of formulas in *Old*. In addition, formulas can be added to the fields *New* and *Next* of either q' or q_1 and q_2, according to the following cases:

- *η is a proposition p, the negation of a proposition $\neg p$, or a boolean constant True or False.* If η is *False*, or $\neg\eta$ is in *Old* (we identify $\neg\neg A$ with A), then the current node q is discarded since it contains a contradiction. This is easy to do; we just return the list *Nodes* without adding q to it.

if $\eta = False$ **or** $Neg(\eta) \in Old(q)$ **then return**(*Nodes*);

Otherwise, the node q is replaced by q' as described below.

$q' := [\ ID \Leftarrow new_ID(),$

$Incoming \Leftarrow Incoming(q)$,
$Old \Leftarrow Old(q) \cup \{\eta\}$,
$New \Leftarrow New(q)$,
$Next \Leftarrow Next(q)]$;
$expand(q', Nodes)$;

- $\eta = \mu \mathbf{U} \psi$. Because $\mu \mathbf{U} \psi$ is equivalent to $\psi \vee (\mu \wedge \mathbf{X}(\mu \mathbf{U} \psi))$, the node q is split into two nodes q_1 and q_2. In node q_1, μ is added to *New* and $\mu \mathbf{U} \psi$ to *Next*. In node q_2, ψ is added to *New*. This is illustrated in Figure 9.10, with $\mu = A$ and $\psi = (B \mathbf{U} C)$.

$q_1 := [ID \Leftarrow new_ID()$,
$Incoming \Leftarrow Incoming(q)$,
$Old \Leftarrow Old(q) \cup \{\eta\}$,
$New \Leftarrow New(q) \cup \{\mu\}$,
$Next \Leftarrow Next(q) \cup \{\mu \mathbf{U} \psi\}]$;
$q_2 := [ID \Leftarrow new_ID()$,
$Incoming \Leftarrow Incoming(q)$,
$Old \Leftarrow Old(q) \cup \{\eta\}$,
$New \Leftarrow New(q) \cup \{\psi\}$,
$Next \Leftarrow Next(q)]$;
$expand(q_2, expand(q_1, Nodes))$;

- $\eta = \mu \mathbf{R} \psi$. The node q is split. Note that $\mu \mathbf{R} \psi$ is equivalent to $\psi \wedge (\mu \vee \mathbf{X}(\mu \mathbf{R} \psi))$, which in turn is equivalent to $(\psi \wedge \mu) \vee (\psi \wedge \mathbf{X}(\mu \mathbf{R} \psi))$. Therefore, ψ is added to *New* of both q_1 and q_2, μ is added to *New* of q_1, and $\mu \mathbf{R} \psi$ is added to *Next* of q_2. The code for this case is similar to the code for $\eta = \mu \mathbf{U} \psi$.
- $\eta = \mu \vee \psi$. Then, q is split. μ is added to *New* of q_1, and ψ is added to *New* of q_2. The code for this case is similar to the code for $\eta = \mu \mathbf{U} \psi$.
- $\eta = \mu \wedge \psi$. Then, q is replaced by q'. Both μ and ψ are added to *New* of q', because the truth of both formulas is needed to make η true.

$q' := [ID \Leftarrow new_ID()$,
$Incoming \Leftarrow Incoming(q)$,
$Old \Leftarrow Old(q) \cup \{\eta\}$,
$New \Leftarrow New(q) \cup \{\mu, \psi\}$,
$Next \Leftarrow Next(q)]$;
$expand(q', Nodes)$;

- $\eta = \mathbf{X} \mu$. Then q is simply replaced by q', and μ is added to *Next* of q'.

$$
\begin{aligned}
q' := [&ID \Leftarrow new_ID(),\\
&Incoming \Leftarrow Incoming(q),\\
&Old \Leftarrow Old(q) \cup \{\eta\},\\
&New \Leftarrow New(q),\\
&Next \Leftarrow Next(q) \cup \{\mu\}];\\
expand(&q', Nodes);
\end{aligned}
$$

The algorithm then recursively expands the new copies. Once a node q is split or is replaced by a new version, it is possible to reclaim the memory used by the node. We will not describe this *garbage collection* process explicitly as part of our algorithm.

The correctness of the algorithm is a consequence of two invariants. Let $\bigwedge Y$ be the conjunction of the formulas in the set Y.

1. When a node q is split into q_1 and q_2, the following invariant holds:

$$
\begin{aligned}
&\Big(\bigwedge Old(q) \wedge \bigwedge New(q) \wedge \mathbf{X} \bigwedge Next(q)\Big)\\
&\quad \longleftrightarrow \Big((\bigwedge Old(q_1) \wedge \bigwedge New(q_1) \wedge \mathbf{X} \bigwedge Next(q_1)) \vee\\
&\qquad (\bigwedge Old(q_2) \wedge \bigwedge New(q_2) \wedge \mathbf{X} \bigwedge Next(q_2))\Big)
\end{aligned}
$$

2. When a node q is replaced by q', the following invariant holds:

$$
\begin{aligned}
&\Big(\bigwedge Old(q) \wedge \bigwedge New(q) \wedge \mathbf{X} \bigwedge Next(q)\Big)\\
&\quad \longleftrightarrow \Big(\bigwedge Old(q') \wedge \bigwedge New(q') \wedge \mathbf{X} \bigwedge Next(q')\Big)
\end{aligned}
$$

The list of nodes *Nodes* constructed by the above algorithm can now be converted into a generalized Büchi automaton with the following components:

- The alphabet Σ consists of sets of propositions from AP. An element $\alpha \in \Sigma$ corresponds to a truth assignment that assigns the value *True* to the propositions in α and *False* to the propositions that are not in α. (In practice, an edge labeled with a boolean expression can be used to represent a set of transitions as shown in Figures 9.3 and 9.4.)
- The set of states Q includes the nodes in *Nodes* and the additional state *init*.
- $(r, \alpha, r') \in \Delta$ if and only if $r \in Incoming(r')$ and α satisfies the conjunction of the negated and nonnegated propositions in $Old(r')$
- The initial state is *init*. There are no incoming edges to *init*.

- The acceptance set F contains a separate set of states $P_i \in F$ for each subformula of the form $\mu \mathbf{U} \psi$; P_i contains all the states r such that either $\psi \in Old(r)$ or $\mu \mathbf{U} \psi \notin Old(r)$. Thus, P_i guarantees that if $\mu \mathbf{U} \psi$ holds at some state in some accepting run, then ψ must hold later on the same run.

The algorithm can be made more efficient by not storing in *Old* the subformulas of the form $\mu \vee \psi$, $\mu \wedge \psi$ and $\mathbf{X}\, \mu$, once they are removed from *New*, unless they form the righthand side of an *until* subformula. It is easy to see that these formulas are redundant. For example, if the field *Old* of some node in the list *Nodes* contains $\mu \vee \psi$, then it also contains either μ or ψ. The other cases are similar.

The number of nodes constructed by the algorithm and the time complexity are exponential in the size of the formula. However, experience shows that the constructed automaton is usually small.

As discussed in Section 9.2, for model checking we need the automaton representing the *bad behaviors*, that is, the ones that are not allowed by the specification φ. Translating φ into an automaton and then complementing it may result in an automaton whose size is doubly exponential in the size of the formula. A much better solution is to translate $\neg\varphi$ into an automaton. At worst, this results in an automaton that is exponentially bigger than φ.

9.5 On-the-Fly Model Checking

In the previous sections, we saw various algorithms that can be combined together for checking whether a system satisfies a property φ. The modeled system is converted into a corresponding Büchi automaton $\mathcal{A}$, and the negation of the specification φ is translated into another automaton $\mathcal{S}$. Then, the emptiness of the intersection of $\mathcal{A}$ and $\mathcal{S}$ is checked. If the intersection is not empty, a counterexample is reported. We will now show how to exploit the machinery developed so far in order to perform the model checking in an efficient way. Instead of constructing the automata for both $\mathcal{A}$ and $\mathcal{S}$ first, we will only construct the property automaton $\mathcal{S}$. We then use it to guide the construction of the system automaton $\mathcal{A}$ while computing the intersection. In this way, we may frequently construct only a small portion of the state space before we find a counterexample to the property being checked.

Explicit state model checking, as described in Chapter 4, uses a graph to represent a Kripke structure with nodes for states and edges for transitions. Extracting this structure from a concurrent system, prior to model checking, may result in a graph that is exponentially bigger than the system.

By using the automata theoretic approach to model checking, as described in this chapter, it is possible in many cases to avoid constructing the entire state space of the modeled system. This is because the states of the automaton $\mathcal{A}$ are generated only when needed,

while checking the emptiness of its intersection with the property automaton $\mathcal{S}$. This tactic is called "on-the-fly" model checking [84, 113].

Thus, one advantage of on-the-fly model checking is that when computing the intersection of the system automaton $\mathcal{A}$ with the property automaton $\mathcal{S}$, some states of $\mathcal{A}$ may never be generated at all. Another advantage of the on-the-fly procedure is that a counterexample may be found before completing the construction of the intersection of the two automata. Once a counterexample has been found and reported, there is no need to complete the construction.

Specifically, suppose that the double DFS of Section 9.3 is used to check the emptiness of the intersection of $\mathcal{A}$ and $\mathcal{S}$. Recall that the states used in constructing the automaton for the intersection are pairs consisting of a state from $\mathcal{A}$ and a state from $\mathcal{S}$. Note that all the states of $\mathcal{A}$ are accepting. Hence, a state of the automaton for the intersection is accepting if and only if its $\mathcal{S}$ component is accepting.

In on-the-fly model checking, the states of the automaton for the intersection are computed as they are needed by the double DFS algorithm. Assume that the automaton $\mathcal{S}$ has already been constructed from $\neg\varphi$. Assume also that part of the automaton $\mathcal{A}$ used in the search so far has already been constructed.

Let $s = \langle r, q\rangle$ be the current state of the search, where r is a state of $\mathcal{A}$ and q a state of $\mathcal{S}$. To continue the search we compute the successors of s one at a time. Because $\mathcal{S}$ is already constructed, the successors $q_1, q_2, \ldots q_n$ of q in $\mathcal{S}$ have already been computed. Let r' be the successor of r that is calculated next. Then, a successor $s_i = \langle r', q_i\rangle$, where $1 \leq i \leq n$, exists exactly if the labelings of the transition from r to r' and from q to q_i with propositions from AP are the same. The two ways of reducing the state space using on-the-fly model checking can now be described.

1. The labeling of r' does not agree with any of the successors q_i of q. Then the search algorithm does not continue to explore the successors of r'.
2. A cycle is detected before the search algorithm backtracks to s. The search then terminates before additional successors of s, which may involve other successors of r, are explored.

In both cases, a reduction in the number of states comes from guiding the construction of $\mathcal{A}$ by the checked property using the automaton $\mathcal{S}$.

9.6 Checking Language Containment Symbolically

In this section we show how symbolic model checking techniques can be used to decide language containment between ω-automata. Although there are many types of ω-automata,

in this chapter we consider only Büchi automata. Algorithms for other types of automata can be derived in a similar fashion from results in [60]. In general, checking language inclusion between two nondeterministic ω-automata is PSPACE-hard. For this reason we consider a restricted case of the general problem in which the specification automaton is deterministic. Thus, our algorithm cannot be used in those cases where the specification cannot be expressed using a deterministic automaton (see Section 9.2.1). For simplicity we also require that both automata are complete.

Let $\mathcal{A} = (\Sigma,\ Q,\ \Delta,\ Q^0,\ F)$ and $\mathcal{A}' = (\Sigma,\ Q',\ \Delta',\ Q^{0'},\ F')$ be two Büchi automata over the same alphabet Σ. Let $M(\mathcal{A}, \mathcal{A}')$ be a Kripke structure $(Q \times Q', R, L)$ over $AP = \{q, q'\}$, where q, q' are two new symbols and

$$q \in L((s, s')) \quad \text{iff } s \in F.$$

$$q' \in L((s, s')) \quad \text{iff } s' \in F'.$$

$$(s, s')R(r, r') \quad \text{iff } \exists \sigma \in \Sigma : (s, \sigma, r) \in \Delta \text{ and } (s', \sigma, r') \in \Delta'.$$

Recall that in Section 5.2 we showed how to encode Kripke structures symbolically. In [60], it is shown that, if $\mathcal{A}'$ is deterministic,

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}') \Leftrightarrow M(\mathcal{A}, \mathcal{A}') \models \mathbf{A}(\mathbf{GF}q \Rightarrow \mathbf{GF}q')$$

Note that the formula above is not a CTL formula, in that there are temporal operators that are not immediately preceded by path quantifiers. However, it is equivalent to **AG AF** q' ("infinitely often q'") under the fairness constraint "infinitely often q." Checking the above formula with the given fairness constraint can be handled using the techniques described in Section 6.2.

THEOREM 8 $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$ if and only if $M(\mathcal{A}, \mathcal{A}') \models$ **AG AF** q' with fairness constraint q.

10 Partial Order Reduction

The *partial order reduction* is aimed at reducing the size of the state space that needs to be searched by model checking algorithms. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. Thus, this reduction technique is best suited for asynchronous systems (in synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved).

The method consists of constructing a reduced state graph. The full state graph, which may be too big to fit in memory, is never constructed. The behaviors of the reduced graph are a subset of the behaviors of the full state graph. The justification of the reduction method shows that the behaviors that are not present do not add any information. More precisely, it is possible to define an equivalence relation among behaviors such that the checked property cannot distinguish between equivalent behaviors. If a behavior is not present in the reduced state graph, then an equivalent behavior must be included.

The name *partial order reduction* has its justification in early versions of the algorithms that were based on the partial order model of program execution [126, 153, 244]. However, the method can be described better as *model checking using representatives* [210, 212], since the verification is performed using representatives from the equivalence classes of behaviors.

In this chapter the *transitions* of a system play a significant role. The partial order reduction is based on the *dependency relation* that exists between the transitions of a system. Furthermore, this reduction method specifies which transitions should be included in the reduced model and which should not. As in Chapter 7, we want to distinguish between different transitions in a system. Thus, we modify the definition of a Kripke structure slightly. Instead of having one transition relation R, we will now have a *set* of transition relations T. For simplicity, we will refer to each element α in T as a *transition*, instead of a transition relation.

A *state transition system* is a quadruple (S, T, S_0, L) where the set of states S, the set of initial states S_0, and the labeling function L are defined as for Kripke structures, and T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$. A Kripke structure $M = (S, R, S_0, L)$ may be obtained by defining R so that $R(s, s')$ holds when there exists a transition $\alpha \in T$ such that $\alpha(s, s')$.

For a transition $\alpha \in T$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is *disabled* in s. The set of transitions enabled in s is $enabled(s)$. A transition α is *deterministic* if for every state s there is at most one state s' such that $\alpha(s, s')$. When α is deterministic we often write $s' = \alpha(s)$ instead of $\alpha(s, s')$. Henceforth, we will only consider deterministic transitions.

A *path* from a state s in a state transition system is a finite or infinite sequence defined as follows. $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that $s = s_0$ and for every i, $\alpha_i(s_i, s_{i+1})$ holds. Here, we do not require paths to be infinite. Moreover, any prefix of a path is also a path. If

π is finite, then the *length* of π is the number of transitions in π and will be denoted by $|\pi|$.

10.1 Concurrency in Asynchronous Systems

A common observation about concurrent asynchronous systems is that the interleaving model imposes an arbitrary ordering between concurrent events. To avoid discriminating against any particular ordering, the events are interleaved in all possible ways. The ordering between independent transitions is largely meaningless. However, common specification languages, including many temporal logics, can distinguish between behaviors that only differ in this manner. Our aim is to take advantage of the cases where the specifications do not distinguish between such behaviors. In these cases, the partial order reduction only checks a subset of the behaviors. However, it checks sufficiently many of them to guarantee the soundness of the verification.

Putting concurrent events in various possible orderings is a potential cause of the state explosion problem. To see this, consider n transitions that can be executed concurrently. In this case, there are $n!$ different orderings and 2^n different states (one state for each subset of the transitions). If the specification does not distinguish between these sequences, it is clearly beneficial to consider only one sequence, with $n + 1$ states. This is demonstrated in Figure 10.1 with $n = 3$.

Our aim is to reduce the number of states that are considered in the model checking process, while preserving the correctness of the checked property. We will assume for

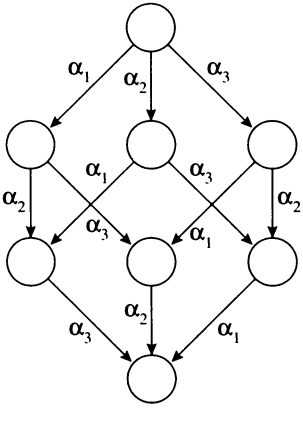


Figure 10.1
Executing three independent transitions.

simplicity of presentation that a *reduced state graph* is first generated explicitly using DFS. The model checking algorithm is then applied to the resulting state graph. The reduction constructs a graph with fewer states and edges. This speeds up the construction of the graph and uses less memory, thus resulting in a more efficient model checking algorithm. Moreover, the reduction can be applied *on-the-fly* while doing the model checking [209]. The DFS can also be replaced by breadth first search [55] and combined with symbolic model checking [4, 164].

The reduction is performed by modifying the DFS used to construct the state graph, as in Figure 10.2. The search starts with an initial state s_0 (line 1) and proceeds recursively. For each state s it selects only a subset $ample(s)$ of the enabled transitions $enabled(s)$ (in line 5), rather than the full set of enabled transitions, as in the full state space construction. The DFS explores only successors generated by these transitions (lines 6–16). In the DFS algorithm in Figure 10.2, a state is labeled as *on_stack* (lines 2,12) when it is first encountered and as *completed* (line 17) when all of its successors have been searched. Thus, a state is marked *on_stack* when it is on the DFS search stack. This information is useful for computing the function *ample*.

```
1   hash(s0);
2   set on_stack(s0);
3   expand_state(s0);

4   procedure expand_state(s)
5         work_set(s) := ample(s);
6         while work_set(s) is not empty do
7               let α ∈ work_set(s);
8               work_set(s) := work_set(s) \ {α};
9               s' := α(s);
10              if new(s') then
11                    hash(s');
12                    set on_stack(s');
13                    expand_state(s');
14              end if;
15              create_edge(s, α, s');
16        end while;
17        set completed(s);
18  end procedure
```

Figure 10.2
Depth-first search with partial order reduction.

When the model checking algorithm is applied to the reduced state graph it terminates with a positive answer when the property holds for the original full state graph. Otherwise, it produces a counterexample. Because the reduced state graph contains fewer behaviors, the counterexample can differ from the one that would have resulted from using the full state graph.

Notice that the algorithm in Figure 10.2 constructs the reduced state graph *directly*. Constructing the full state graph and later reducing it would defy the purpose of the reduction.

In order to implement the algorithm we must find a systematic way of calculating $ample(s)$ for any given state s. The calculation of $ample(s)$ needs to satisfy three goals:

1. When $ample(s)$ is used instead of $enabled(s)$, sufficiently many behaviors must be present in the reduced state graph so that the model checking algorithm gives correct results.
2. Using $ample(s)$ instead of $enabled(s)$ should result in a significantly smaller state graph.
3. The overhead in calculating $ample(s)$ must be reasonably small.

10.2 Independence and Invisibility

In this section, we will define two concepts that can assist in reducing the state graph. As noted earlier, in the interleaving model for concurrent systems, transitions that can be executed concurrently from some state are interleaved in either order. This can be formulated by defining an independence relation on pairs of transitions that can execute concurrently. An *independence* relation $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:

Enabledness If $\alpha,\ \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.

Commutativity $\alpha,\ \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The *dependency* relation D is the complement of I, namely

$$D = (T \times T) \setminus I.$$

The enabledness condition states that a pair of independent transitions do not *disable* one another. Note, however, that it is possible for one to *enable* another. Note that the definition makes use of the fact that I is symmetric. The commutativity condition, which is well defined due to the enabledness condition, states that executing independent transitions in either order results in the same state. These conditions are illustrated in Figure 10.3.

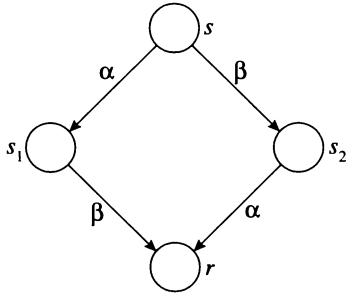


Figure 10.3
Execution of independent transitions.

When it is hard to check whether two transitions α and β are independent or not, assuming that they are dependent always preserves the correctness of the reductions described in this chapter.

The definition of independence can be used for the reduction even when two independent transitions cannot actually be executed in parallel. For example, when two transitions of different processes increment a shared variable, they satisfy the independence conditions, although some type of physical arbitration must be used to prevent them from executing simultaneously.

The commutativity condition, illustrated in Figure 10.3, suggests a potential reduction to the state graph, for it does not matter whether α is executed before β or vice versa in order to reach the state r from s. Thus, it is tempting to select only one of the transitions originating from s. This is not appropriate for the following reasons:

PROBLEM 1 The checked property might be sensitive to the choice between the states s_1 and s_2, not only the states s and r.

PROBLEM 2 The states s_1 and s_2 may have other successors in addition to r, which may not be explored if either is eliminated.

We will return to these problems at the end of Section 10.3. The first step in solving them is to define what it means for a transition to be invisible.

Let $L : S \rightarrow 2^{AP}$ be the function that labels each state with a set of atomic propositions. A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) \cap AP' = L(s') \cap AP'$. In other words, a transition is invisible when its execution from any state does not change the value of the propositional variables in AP'. A transition is *visible* if it is not invisible.

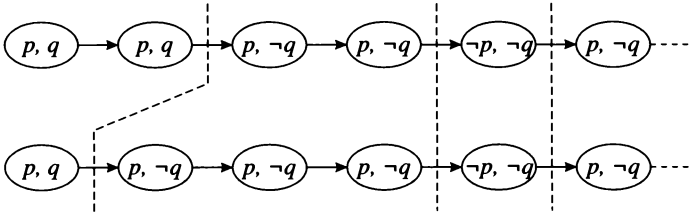


Figure 10.4
Two stuttering equivalent paths.

A closely related concept is that of *stuttering* [167], which refers to a sequence of identically labeled states along a path in a Kripke structure. Two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are *stuttering equivalent* (see Figure 10.4), denoted $\sigma \sim_{st} \rho$ if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = \cdots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \cdots = L(r_{j_{k+1}-1}).$$

We call a finite sequence of identically labeled states a *block*. Intuitively, two paths are stuttering equivalent when they can be partitioned into infinitely many blocks, such that the states in the kth block of one are labeled the same as the states in the kth block of the other. Note that corresponding blocks may have different lengths. Stuttering equivalence can be defined in a similar way for finite paths using finite sequences of indexes $0 = i_0 < i_1 < i_2 < \ldots i_n$ and $0 = j_0 < j_1 < j_2 < \ldots j_n$. Stuttering is a particularly important concept for asynchronous systems because there is no correlation between the time separating two events and the number of transitions occurring between them.

An LTL formula $\mathbf{A}f$ is *invariant under stuttering* if and only if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$,

$$\pi \models f \quad \text{if and only if} \quad \pi' \models f.$$

We denote the subset of the logic LTL without the next time operator by LTL_{-X}.

THEOREM 9 Any LTL_{-X} property is invariant under stuttering.

The theorem is proved using a simple induction on the size of the LTL formula. It is interesting to note that the converse of Theorem 9 also holds [211]:

THEOREM 10 Every LTL property that is stuttering closed can be expressed in LTL_{-X}.

We now extend the notion of stuttering equivalence to structures. Two structures M and M' are *stuttering equivalent* if and only if

- M and M' have the same set of initial states.
- For each path σ of M that starts from an initial state s of M there exists a path σ' of M' from the same initial state s such that $\sigma \sim_{st} \sigma'$, and
- for each path σ' of M' that starts from an initial state s of M' there exists a path σ of M from the same initial state s such that $\sigma' \sim_{st} \sigma$.

The following corollary is useful for showing that an LTL_{-X} formula does not distinguish between structures that are stuttering equivalent. It will be exploited later, for the partial order reduction generates a structure that is stuttering equivalent to the full state graph.

COROLLARY 2 Let M and M' be two stuttering equivalent structures. Then, for every LTL_{-X} property $\mathbf{A}f$, and every initial state $s \in S_0$, $M,\ s \models \mathbf{A}f$ if and only if $M',\ s \models \mathbf{A}f$.

Returning to Figure 10.3, suppose that at least one transition, say α, is invisible, then $L(s) = L(s_1)$ and $L(s_2) = L(r)$. Consequently,

$s\ s_1\ r \sim_{st} s\ s_2\ r$

10.3 Partial Order Reduction for LTL_{-X}

When the specification is invariant under stuttering, commutativity and invisibility allow us to avoid generating some of the states. Based on this observation, we suggest a systematic way of selecting an ample set of transitions for any given state. The ample sets will be used by the DFS algorithm to construct a reduced state graph so that for every path not considered by the DFS algorithm there is a stuttering equivalent path that is considered. This guarantees that the reduced state graph is stuttering equivalent to the full state graph.

We say that state s is *fully expanded* when $ample(s) = enabled(s)$. In this case, all of the successors of that state will be explored by the DFS algorithm.

Instead of giving a specific algorithm for constructing ample sets, we will first provide four conditions for selecting $ample(s) \subseteq enabled(s)$ such that the satisfaction of the LTL_{-X} specification is preserved. The reduction will depend on the set of propositions AP' that appear in the LTL_{-X} formula.

Condition **C0** guarantees that if the state has at least one successor, then the reduced state graph also contains a successor for this state.

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.

Condition **C1** is the most complicated among the constraints on $ample(s)$.

C1 [126, 153, 208, 244] Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

Note that Condition **C1** refers to paths in the *full* state graph. We need a way of checking that **C1** holds without actually constructing the full state graph. Later, we will show how to restrict **C1** so that $ample(s)$ can be calculated based on the current state s.

LEMMA 24 The transitions in $enabled(s) \setminus ample(s)$ are all independent of those in $ample(s)$.

Proof Let $\gamma \in enabled(s) \setminus ample(s)$. Suppose that $(\gamma, \delta) \in D$, where $\delta \in ample(s)$. Because γ is enabled in s, in the full graph there is a path starting with γ. But then a transition dependent on some transition in $ample(s)$ is executed before a transition in $ample(s)$, contradicting Condition **C1**. □

In order to guarantee the correctness of the DFS reduction algorithm, we need to know that if we always choose the next transition to explore from $ample(s)$, we do not omit any paths that are essential for checking the correctness of the state graph. Condition **C1** implies that such a path will have one of two forms:

- The path has a prefix $\beta_0\beta_1 \ldots \beta_m\alpha$, where $\alpha \in ample(s)$ and each β_i is independent of all transitions in $ample(s)$ including α.
- The path is an infinite sequence of transitions $\beta_0\beta_1 \ldots$ where each β_i is independent of all transitions in $ample(s)$.

Condition **C1** also implies that, if along a finite sequence of transitions $\beta_0\beta_1 \ldots \beta_m$ executed from s, none of the transitions in $ample(s)$ have occurred, then all the transitions in $ample(s)$ remain enabled. This is because each β_i is independent of the transitions in $ample(s)$ and, therefore, cannot disable them.

In the first case, assume that the sequence of transitions $\beta_0\beta_1 \ldots \beta_m\alpha$ reaches a state r. This sequence will not be considered by the DFS algorithm. However, by applying the enabledness and commutativity conditions m times, we can construct a finite sequence $\alpha\beta_0\beta_1 \ldots \beta_m$, that also reaches r. This is illustrated in Figure 10.5. In other words, even if the reduced state graph does not contain the sequence $\beta_0\beta_1 \ldots \beta_m\alpha$ that reaches the state r, we can still construct from s another sequence that reaches the same state r.

Consider the two sequences of states $\sigma = s_0s_1 \ldots s_mr$ and $\rho = sr_0r_1 \ldots r_m$ in Figure 10.5, generated by $\beta_0\beta_1 \ldots \beta_m\alpha$ and $\alpha\beta_0\beta_1 \ldots \beta_m$, respectively. In order to discard σ, we want σ and ρ to be stuttering equivalent. This is guaranteed if α is invisible, for then

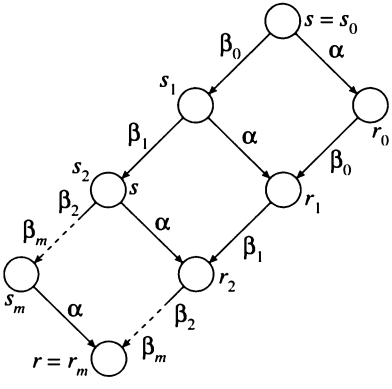


Figure 10.5
Transition α commutes with $\beta_0\beta_1 \ldots \beta_m$.

$L(s_i) = L(r_i)$ for $0 \leq i \leq m$. Thus, the checked property will not be able to distinguish between the two sequences above. This can be achieved by condition **C2**:

C2 [Invisibility [209]] If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.

Consider now the second case, in which an infinite path $\beta_0\beta_1\beta_2 \ldots$ that starts at s does not include any transition from $ample(s)$. By Condition **C2** all transitions in $ample(s)$ are invisible. Let α be such a transition in $ample(s)$, then the path generated by the infinite sequence of transitions $\alpha\beta_0\beta_1\beta_2 \ldots$ is stuttering equivalent to the one generated by $\beta_0\beta_1\beta_2 \ldots$. Again, even though the path $\beta_0\beta_1\beta_2 \ldots$ is not included in the reduced state graph, there is a stuttering equivalent path that is included.

Conditions **C1** and **C2** are not yet sufficient to guarantee that the reduced state graph is stuttering equivalent to the full state graph. In fact, there is a possibility that some transition will actually be delayed forever because of a cycle in the constructed state graph. As an example, consider the processes in Figure 10.6. Assume that the transition β is independent of the transitions α_1, α_2, and α_3. The transitions α_1, α_2, and α_3 are interdependent. The process on the left can execute the visible transition β exactly once. Assume there is one proposition p, which is changed from *True* to *False* by β, so that β is visible. The process on the right performs the invisible transitions α_1, α_2, and α_3 repeatedly in a loop.

The full state graph of the system in Figure 10.6 is shown on the left in Figure 10.7. The right side of the figure shows the first stages of constructing the reduced state graph, where α_1, α_2, and α_3 are invisible. Starting with the initial state s_1, we can select $ample(s_1) = \{\alpha_1\}$. Conditions **C0**, **C1**, and **C2** are satisfied. Thus, we generate $s_2 = \alpha_1(s_1)$. Similarly, we can select $ample(s_2) = \{\alpha_2\}$, generating $s_3 = \alpha_2(s_2)$. Finally, reaching s_3, Conditions **C0**, **C1**,

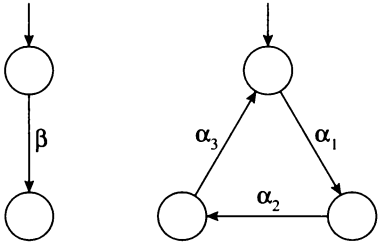


Figure 10.6
Two concurrent processes.

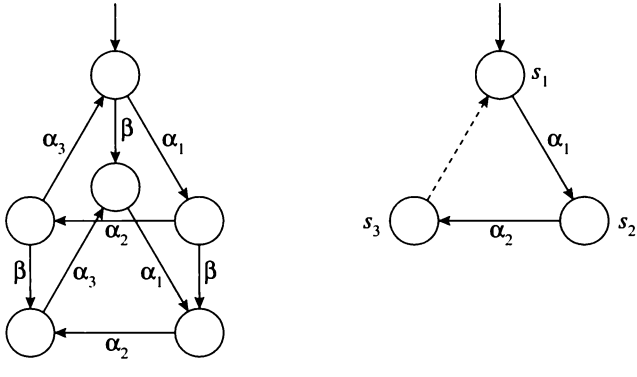


Figure 10.7
Full and reduced state graph.

and **C2** allow selecting $ample(s_3) = \{\alpha_3\}$. But the reduced state graph generated in this way does not contain any sequences where p is changed from *True* to *False*. The problem is that each state along the cycle s_1, s_2, s_3, s_1 has deferred β to a possible future state. When the cycle is closed, the construction terminates, and transition β is ignored.

To remedy this problem, we add the following condition:

C3 [Cycle condition [21, 55, 208]] A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

We are now able to address Problems 1 and 2 described in the previous section. Consider Figure 10.3 again. Assume that the DFS reduction algorithm chooses β as $ample(s)$ and does not include state s_1 in the reduced graph.

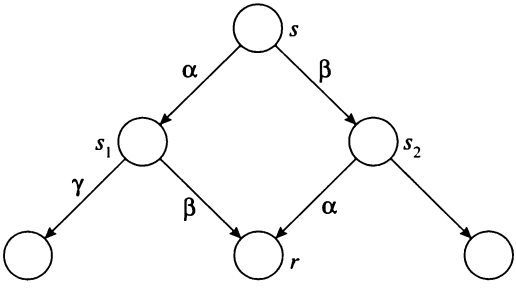


Figure 10.8
Diagram illustrating Problem 2.

We consider Problem 1 first. By Condition **C2**, β must be invisible, thus s, s_2, r and s, s_1, r are stuttering equivalent. In this chapter we are only interested in properties that are invariant under stuttering. Such properties will not be able to distinguish between the two sequences.

We next consider Problem 2. Assume that there is a transition γ enabled from s_1, as in Figure 10.8. We show that γ is still enabled at state r. Moreover, the transition sequences α, γ and β, α, γ lead to stuttering equivalent state sequences. We first note that γ cannot be dependent on β. Otherwise, the sequence α, γ violates Condition **C1**, since a transition dependent on β is executed before β. Thus, γ is independent of β. Because it is enabled in s_1, it must also be enabled in state r. Assume that γ, when executed from r, results in state r' and when executed from s_1 results in state s_1'. Since β is invisible, the two state sequences s, s_1, s_1' and s, s_2, r, r' are stuttering equivalent. Therefore, properties that are invariant under stuttering will not distinguish between the two.

10.4 An Example

Consider the mutual exclusion program P, presented in Chapter 2. The state graph for P is given in Figure 10.9. The states of the program are labeled with $AP = \{NC_i, CR_i, l_i, turn = i, \perp \mid i = 0, 1\}$, where $CR_i \in L(s)$ if $pc_i = CR_i$ in the state s, and $CR_i \notin L(s)$ if $pc_i \neq CR_i$ in s. The labeling $L(s)$ is defined similarly for all other atomic propositions in AP.

Let $f = \mathbf{G}\,\neg(CR_0 \wedge CR_1)$ be an LTL_{-X} formula describing the mutual exclusion property. We will show how the DFS algorithm of Figure 10.2 can be used to construct a reduced state graph that is stuttering equivalent to the full state graph with respect to a

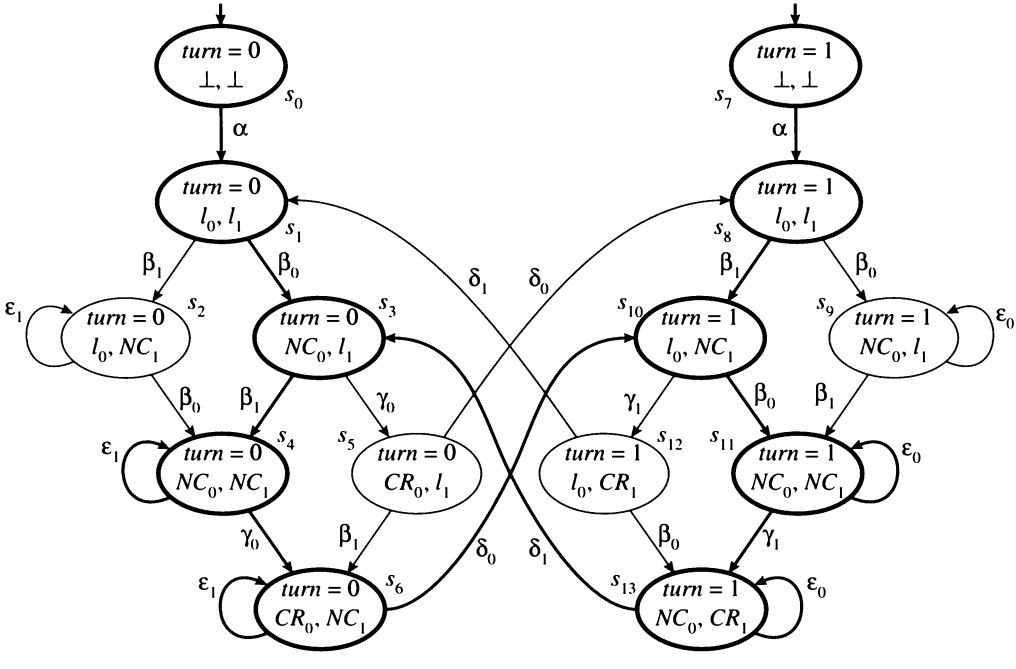


Figure 10.9
Reduced state graph for a mutual exclusion program.

subset AP' of the atomic propositions. Because we are interested in checking whether P satisfies f, we choose $AP' = \{CR_0, CR_1\}$.

Following is a list of the transitions of the program P that are enabled in some reachable state of P, where $i = 0, 1$. For brevity we omitted $same(pc_j)$ for $j \neq i$ from each of the transitions.

α: $pc = m \ \wedge \ pc'_0 = l_0 \ \wedge \ pc'_1 = l_1 \ \wedge pc' = \perp$

β_i: $pc_i = l_i \ \wedge \ pc'_i = NC_i \ \wedge \ True \ \wedge \ same(turn)$

γ_i: $pc_i = NC_i \ \wedge \ pc'_i = CR_i \ \wedge \ turn = i \ \wedge \ same(turn)$

δ_i: $pc_i = CR_i \ \wedge \ pc'_i = l_i \ \wedge \ turn' = (i + 1) \ mod \ 2$

ε_i: $pc_i = NC_i \ \wedge \ pc'_i = NC_i \ \wedge \ turn \neq i \ \wedge \ same(turn)$

The visible transitions with respect to AP' are those in which CR_0 or CR_1 has different values before and after the transition. Thus, $\{\gamma_0, \gamma_1, \delta_0, \delta_1\}$ are visible.

Each transition is dependent on itself because the dependency relation is reflexive. All of the transitions are dependent on α since it must be executed before any other transition in the program. The dependency relation for the remaining transitions is calculated using the following two rules:

- Two transitions that change the same variable (including the program counters) are dependent.
- If one transition sets a variable and the other checks that variable, then the transitions are dependent.

Thus, all of the transitions in the same process are interdependent. Also, (γ_1, δ_0), (γ_0, δ_1), $(\varepsilon_1, \delta_0)$, $(\varepsilon_0, \delta_1)$, (δ_0, δ_1) are in D since δ_i changes the variable *turn*, while γ_i and ε_i check its value. Finally, we complete the relation D to be symmetric.

Figure 10.9 shows the full state graph. The states and edges included in the reduced state graph are shown using thick lines. Following are the states of the reduced state graph in the order they are visited by the DFS algorithm: $s_0, s_1, s_3, s_4, s_6, s_{10}, s_{11}, s_{13}, s_7, s_8$.

The DFS algorithm starts with s_0, which is one of the two initial states. For this state, $ample(s_0) = enabled(s_0) = \{\alpha\}$. For s_1, it is possible to select as $ample(s_1)$ either $\{\beta_0\}$, $\{\beta_1\}$ or $\{\beta_0, \beta_1\}$. The latter will usually result in a smaller reduction and therefore will not be considered. The first choice corresponds to selecting the enabled transitions of P_0, whereas the second choice corresponds to selecting P_1. Condition **C0** is trivially satisfied. In both cases, **C1** is satisfied. For example, suppose $ample(s_1) = \{\beta_0\}$ then along all paths leaving s_1, either β_0 is immediately executed or β_1 is executed before β_0. However, β_1 is independent of β_0.

Condition **C2** is also satisfied, for β_0 and β_1 are invisible. Finally, **C3** is satisfied because no cycle is yet formed. The choice between the two sets is arbitrary, although one may provide a better reduction in a later stages of the algorithm. We select $ample(s_1) = \{\beta_0\}$.

Executing β_0 from s_1 results in the state s_3. By using a similar argument, we select as $ample(s_3)$, the transitions of P_1 that are enabled in s_3, namely $\{\beta_1\}$. Next, we select $ample(s_4) = \{\gamma_0, \varepsilon_1\}$. We cannot select for s_4 the set $\{\gamma_0\}$, since γ_0 is visible. We cannot also select the singleton $\{\varepsilon_1\}$, because this will construct a self loop on which the transition γ_0 is enabled but never included in an ample set, thus violating Condition **C3**.

We can now select, $ample(s_6) = \{\varepsilon_1, \delta_0\}$. Because they are dependent we have to choose both in order not to violate Condition **C1**. For states s_{10} and s_{11} we choose $ample(s_{10}) = \{\beta_0\}$ and $ample(s_{11}) = \{\gamma_1, \varepsilon_0\}$. The arguments are similar to the ones for states s_3 and s_4, respectively. We next select $ample(s_{13}) = \{\delta_1, \varepsilon_0\}$. The transition δ_1 taken from s_{13} closes the cycle $s_3\ s_4\ s_6\ s_{10}\ s_{11}\ s_{13}$. By examining Figure 10.9 it is easy to check that Condition **C3** is satisfied for this cycle.

The DFS algorithm continues the search from the other initial state s_7. We select $ample(s_7) = \{\alpha\}$. Based on arguments similar to those for s_1, we also select $ample(s_8) = \{\beta_1\}$. By executing β_1 from s_8, we reach only the state s_{10} that has already been visited. Thus, the algorithm terminates.

A model-checking algorithm for LTL can now be applied to check if the reduced state graph constructed by the algorithm satisfies the formula f because $f \in \text{LTL}_{-X}$. The full state graph satisfies the formula if and only if the reduced state graph does.

10.5 Calculating Ample Sets

10.5.1 The Complexity of Checking the Conditions

In order to make the partial order reduction efficient, we need to be able to calculate the ample sets for the states in the reduced graph with minimal overhead. We will consider the related problem of checking Conditions **C0** to **C3** for a set of enabled transitions at a given state. Condition **C0** for a particular state can be checked in constant time. Condition **C2** is also simple to check, by examining the transitions in the set.

Condition **C1** is a constraint that is not immediately checkable by examining the current state of the search, in that it refers to future states (some of which need not even be in the reduced state graph). The next theorem shows that, in general, checking **C1** is at least as hard as searching the full state space.

THEOREM 11 Checking Condition **C1** for a state s and a set of transitions $T \subseteq enabled(s)$ is at least as hard as checking REACHABILITY for the full state space.

Proof Consider checking whether a state r is reachable in a transition system $\mathcal{T}$ from an initial state s_0. We will reduce this problem to deciding condition **C1**. First, let α and β be new transitions. Let the transition α be only enabled at the state r. Let the transition β be enabled from the initial state and independent of all the transitions of $\mathcal{T}$. We construct β and α so that they are dependent (e.g., they both change the value of the same variable).

Consider $\{\beta\}$ as a candidate for being an ample set from s_0. First assume that **C1** is violated. Then there is a path in the new state graph along which α is performed before β. Because α is enabled only in r, this path leads from s_0 to r. The sequence of transitions on the path from s_0 to r exists also in the original state graph, in that it does not include the added transitions α or β. Thus r is reachable from s_0 in the original system.

For the other direction, assume that r is reachable in the original state graph from s_0. Then, there is a sequence from s_0 to r, which does not include β. This sequence also appears in the new state graph, and now can be extended by the transition α taken from r. The resulting sequence violates **C1**. □

In view of the previous theorem, we will avoid checking Condition **C1** for an arbitrary subset of enabled transitions. In Section 10.5.2 we will give a procedure to compute a set of transitions that is guaranteed by construction to satisfy **C1**. Although the procedure may not lead to ample sets that achieve the greatest possible reduction, it is quite efficient. There is evidently a tradeoff between efficiency of computation and the amount of reduction.

Condition **C3** is also defined in global terms. However, it refers to the reduced state graph, whereas **C1** refers to the full state graph. A possible way of implementing this constraint is to first generate a reduced state graph and then to *correct* it by adding additional transitions until it satisfies **C3** [244]. On the other hand, the approach we take replaces **C3** by a stronger condition that can be checked directly on the current state.

LEMMA 25 A sufficient condition for **C3** is that at least one state along each cycle is fully expanded.

Proof Assume there is a cycle with a fully expanded state, but the cycle does not satisfy Condition **C3**. Thus, we have some transition α that is enabled in some state s of the cycle but is never included in an ample set along the cycle. By lemma 24, if α is not included in an ample set then it is independent of all the transitions in it. Thus, α is independent of all transitions in the ample sets selected along the cycle. Consequently, it remains enabled in all the states along the cycle. However, if one of the states s' is fully expanded, meaning that $ample(s') = enabled(s')$, α is necessarily included in $ample(s')$. This contradicts the assumption that α is never selected. □

Efficient ways of enforcing **C3** are based on the specific search strategy that is used to generate the reduced state space. For depth first search, we can use the fact that every cycle includes an edge that goes back to a node on the search stack. Such an edge is also called a *back edge*. Thus, we strengthen **C3** in the following manner.

C3′ If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search stack.

We thus always try to select an ample set that does not include a back edge. If we do not succeed, the current state is fully expanded.

In breadth first search, the search progresses in levels, where level k consists of a set of states reachable from the initial states using k transitions. A necessary condition for closing a cycle during breadth first search is the following: A transition applied to a state s in the current level results in a state in the current or previous level of the breadth first search. This condition is not sufficient. Consequently, using this condition to detect when a cycle is closed may cause more states than necessary to be fully expanded.

10.5.2 Heuristics for Ample Sets

In view of the complexity results in Section 10.5.1 we give some heuristics for calculating ample sets. The algorithm will depend on the model of computation. We will consider shared variables and message passing with handshaking and with queues.

Common to all of these models of computation is the notion of a *program counter*, which is part of the state. We will denote the program counter of a process P_i in a state s by $pc_i(s)$.

In order to present the algorithm, we will use the following notation:

- $pre(\alpha)$ is a set of transitions that includes the transitions whose execution may enable α. More formally, $pre(\alpha)$ includes all the transitions β such that there exists a state s for which $\alpha \notin enabled(s)$, $\beta \in enabled(s)$, and $\alpha \in enabled(\beta(s))$.
- $dep(\alpha)$ is the set of transitions that are dependent on α, that is,

$\{\beta | (\beta, \alpha) \in D\}$.

- T_i is the set of transitions of process P_i. $T_i(s) = T_i \cap enabled(s)$ denotes the set of transitions of P_i that are enabled in the state s.
- $current_i(s)$ is the set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$. The set $current_i(s)$ always contains $T_i(s)$. In addition, it may include transitions whose program counter has the value $pc_i(s)$, but are not enabled in s.

Note that on any path starting from s, some transition in $current_i(s)$ must be executed before other transitions of T_i can execute. The definitions of $pre(\alpha)$ and the dependency relation D (which directly effects $dep(\alpha)$) may not be exact. The set $pre(\alpha)$ may contain transitions that do not enable α. Likewise, the dependency relation D may also include pairs of transitions that are actually independent. This freedom makes it possible to calculate ample sets efficiently while still preserving the correctness of the reduction.

The above definitions are extended to sets in the natural way. For instance, $dep(T) = \cup_{\alpha \in T} dep(\alpha)$.

Next, we specialize $pre(\alpha)$ for various models of computation. Recall that $pre(\alpha)$ includes all transitions whose execution from some state can enable α. We construct $pre(\alpha)$ as follows:

- The set $pre(\alpha)$ includes the transitions of the processes that contain α and that can change the program counter to a value from which α can execute.
- If the enabling condition for α involves shared variables then $pre(\alpha)$ includes all other transitions that can change these shared variables.

- If α involves message passing with queues, that is, α sends or receives data on some queue q, then $pre(\alpha)$ includes the transitions of other processes that receive or send data, respectively, through q.

We now describe the dependency relation for the different models of computation.

1. Pairs of transitions that share a variable, which is changed by at least one of them, are dependent.

2. Pairs of transitions belonging to the same process are dependent. This includes in particular pairs of transitions in $current_i(s)$ for any given state s and process P_i. Note that a transition that involves handshaking or rendezvous communication as in CSP or ADA can be treated as a joint transition of both processes. Therefore, it depends on all of the transitions of both processes.

3. Two send transitions that use the same message queue are dependent. This is because executing one may cause the message queue to fill, disabling the other. Also, the contents of the queue depends on their order of execution. Similarly, two receive transitions are dependent.

Note that a pair of send and receive transitions in different processes, which use the same message queue are independent. This is because any one of these transitions can potentially enable the other but can not disable it.

An obvious candidate for $ample(s)$ is the set $T_i(s)$ of transitions enabled in s for some process P_i. Because the transitions in $T_i(s)$ are interdependent, an ample set for s must include either all of the transitions or none of them. To construct an ample set for the current state s, we start with some process P_i such that $T_i(s) \neq \emptyset$. We want to check whether $ample(s) = T_i(s)$ satisfies Condition **C1**. There are two cases in which this selection might violate **C1**. In both of these cases, some transitions independent of those in $T_i(s)$ are executed, eventually enabling a transition α that is dependent on $T_i(s)$. The independent transitions in the sequence cannot be in T_i, since all the transitions of P_i are interdependent.

1. In the first case, α belongs to some other process P_j. A necessary condition for this to happen is that $dep(T_i(s))$ includes a transition of process P_j. By examining the dependency relation, this condition can be checked effectively.

2. In the second case, α belongs to P_i. Suppose that the transition $\alpha \in T_i$ which violates **C1** is executed from a state s'. The transitions executed on the path from s to s' are independent of $T_i(s)$ and hence, are from other processes. Therefore, $pc_i(s') = pc_i(s)$. So α must be in $current_i(s)$. In addition, $\alpha \notin T_i(s)$, otherwise it does not violate **C1**. Thus, $\alpha \in current_i(s) \setminus T_i(s)$.

Since α is not in $T_i(s)$, it is disabled in s. Therefore, a transition in $pre(\alpha)$ must be included in the sequence from s to s'. A necessary condition for this case is that $pre(current_i(s) \setminus T_i(s))$ includes transitions of processes other than P_i. This condition can also be checked effectively.

In both cases we discard $T_i(s)$ as an ample set, and can try the transitions $T_j(s)$ of another process j as a candidate for $ample(s)$. Note that we take a conservative approach discarding some ample sets even though at run-time it might be that Condition **C1** would actually not be violated.

The following code checks Condition **C1** for the enabled transitions of a process P_i, as explained above.

```
function check_C1(s, P_i)
    for all P_j ≠ P_i do
        if dep(T_i(s)) ∩ T_j ≠ ∅
                    or pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅ then
            return False;
        end if;
    end for all;
    return True;
end function
```

The function $check_C2$ is given a set of transitions and returns *True* if all of the transitions in the set are invisible. Otherwise, it returns *False*.

```
function check_C2(X)
    for all α ∈ X do
        if visible(α) then return False;
    return True;
end function
```

The procedure $check_C3'$ tests whether the execution of a transition in a given set $X \subseteq enabled(s)$ is still on the search stack. For that, we can use our marking of the states as on_stack or $completed$ in Figure 10.2. Recall that a state is on_stack when the state is on the search stack.

```
function check_C3'(s, X)
    for all α ∈ X do
        if on_stack(α(s)) then return False;
    return True;
```

end function

The algorithm for $ample(s)$ tries to find a process P_i such that $T_i(s)$ satisfies all the conditions **C0** to **C3**. If no such process can be found, $ample$ returns the set $enabled(s)$.

function $ample(s)$
  for all P_i **such that** $T_i(s) \neq \emptyset$ **do**
    if $check_C1(s, P_i)$ **and** $check_C2(T_i(s))$
      and $check_C3'(s, T_i(s))$ **then**
      return $T_i(s)$;
    end if;
  end for all;
  return $enabled(s)$;
end function

The SPIN [138, 140] system includes an implementation [139] of the partial order reduction. The heuristics used for selecting ample sets are similar to the ones described in this section. However, in SPIN, for many of the states, Conditions **C0**, **C1**, and **C2** are precomputed when the system being verified is translated into its internal representation.

10.5.3 On-the-Fly Reduction

In previous sections of this chapter, the model-checking algorithm was explained as a two-phase process. The reduced state-space is constructed in the first phase. In the second phase, an LTL model-checking algorithm is used to check the correctness of a formula in the reduced state graph. In practice, many model checkers work in a more efficient manner. They combine the construction of the state graph with checking that it satisfies the specification. As shown in Section 9.5, it is frequently possible to identify on-the-fly that the system violates the specification before completing the construction of the state graph. The partial order reduction can be used in conjunction with on-the-fly model checking.

The only condition that needs special attention is the cycle closing Condition **C3**. The cycles in the product of the state graph and the property automaton are not necessarily the same as the ones in the reduced state graph generated in the off-line algorithm. To see this, observe that each state $\langle s, q \rangle$ in the product is a pair of a system state s and a state q of the property automaton. Assume that a cycle is closed at state s in the state graph. In the product, the state s may be paired with a different component of the automaton when it is encountered the second time. Thus, it cannot close a cycle. However, it can be shown [209] that it is correct to check Condition **C3'** with respect to cycles of the product. Intuitively, the purpose of **C3'** is to avoid postponing the inclusion of some transitions forever in the

reduced graph. This is still guaranteed when **C3'** is applied to the cycles of the product. A formal proof appears in [209].

A subtle point arises when the double DFS procedure described in Section 9.3 is used with the partial order reduction. In this case, the order in which the graph is traversed may differ in the first and second phases of the search. As a consequence cycles may be closed at different states in the two phases. Thus, some additional information must be propagated between the two phases, to ensure that the same ample sets will be chosen in both [141].

10.6 Correctness of the Algorithm

Let M be the full state graph of some system. Let M' be a reduced state graph constructed using the partial order reduction algorithm described in Section 10.1.

A *string* is a sequence of transitions from T. Let T^* be the set of all the strings over T. Denote by $vis(v)$, where v is either finite or infinite string, the projection of v onto the visible transitions. Thus, if a and b are visible and c and d are not, then $vis(abddbcbaac) = abbbaa$. Let $tr(\sigma)$ be the sequence of transitions on a path σ. Let v, w be two finite strings. We write $v \sqsubset w$ if v can be obtained from w by erasing one or more transitions. For example $abbcd \sqsubset aabcbccde$. We denote $v \sqsubseteq w$ if either $v = w$ or $v \sqsubset w$.

Let $\sigma \circ \eta$ denote the concatenation of the paths σ and η of M, where σ is finite, and the last state $last(\sigma)$ of σ is the same as the first state $first(\eta)$ of η. The *length* of a path σ, denoted $|\sigma|$, is the number of edges of σ.

Let σ be some infinite path of the full state graph M, starting with some initial state. We will construct an infinite sequence of paths $\pi_0, \pi_1, \ldots,$ where $\pi_0 = \sigma$. Each path π_i will be decomposed into $\eta_i \circ \theta_i$, where η_i is of length i. Assuming that we have constructed the paths $\pi_0, \ldots, \pi_i$, we describe how to construct $\pi_{i+1} = \eta_{i+1} \circ \theta_{i+1}$. Let $s_0 = last(\eta_i) = first(\theta_i)$ and α the transition labeling the first edge of θ_i. Denote

$$\theta_i = s_0 \xrightarrow{\alpha_0=\alpha} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots.$$

There are two cases:

A. $\alpha \in ample(s_0)$. Then select $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\alpha} \alpha(s_0))$. θ_{i+1} is $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$, that is, θ_i without its first edge.

B. $\alpha \notin ample(s_0)$. By **C2**, all of the transitions in $ample(s_0)$ must be invisible since s_0 is not fully expanded. Here again, there are two cases, **B1** and **B2**:

B1. Some $\beta \in ample(s_0)$ appears on θ_i after some sequence of independent transitions $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$, that is, $\beta = \alpha_k$. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} \beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2} \xrightarrow{\alpha_{k+2}} \ldots$ in M. That is, β is moved to appear before $\alpha_0\alpha_1\alpha_2 \ldots \alpha_{k-1}$. Note that $\beta(s_k) = s_{k+1}$. Therefore, $\beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2}$ is the same as $s_{k+1} \xrightarrow{\alpha_{k+1}} s_{k+2}$.

B2. Some $\beta \in ample(s_0)$ is independent of all the transitions that appear on θ_i. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$ in M. That is, β is executed from s_0 and then applied to each state of θ_i.

In both cases $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\beta} \beta(s_0))$ and θ_{i+1} is the path that is obtained from ξ by removing the first transition $s_0 \xrightarrow{\beta} \beta(s_0)$.

Let η be the path such that the prefix of length i is η_i. The path η is well defined in that η_i is constructed from η_{i-1} by appending a single transition.

LEMMA 26 The following hold for all i, j such that $j \geq i \geq 0$.

1. $\pi_i \sim_{st} \pi_j$.
2. $vis(tr(\pi_i)) = vis(tr(\pi_j))$.
3. Let ξ_i be a prefix of π_i and ξ_j be a prefix of π_j such that $vis(tr(\xi_i)) = vis(tr(\xi_j))$. Then $L(last(\xi_i)) = L(last(\xi_j))$.

Proof It is sufficient to consider the case where $j = i + 1$. Consider the three ways of constructing π_{i+1} from π_i. In case **A**, $\pi_i = \pi_{i+1}$, and all three parts of the lemma hold trivially.

Next, consider case **B1** of the construction, in which π_{i+1} is obtained from π_i by executing some invisible transition β in π_{i+1} earlier than it is executed in π_i. In this case, we replace the sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} s_{k-1} \xrightarrow{\beta} s_k$ by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} \beta(s_{k-1})$. Because β is invisible, corresponding states have the same label, that is, for each $0 < l \leq k$, $L(s_l) = L(\beta(s_l))$. Also, the order of the visible transitions remains unchanged. Parts 1, 2, and 3 follow immediately.

Finally, in case **B2** of the construction, the difference between π_i and π_{i+1} is that π_{i+1} includes an additional invisible transition β. Thus, we replace some suffix $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of π_i by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$. So, $L(s_l) = L(\beta(s_l))$ for $l \geq 0$. Again, the order of the visible transitions remains unchanged. As in the previous case, parts 1, 2, and 3 follow immediately. □

LEMMA 27 Let η be the path constructed as the limit of the finite paths η_i. Then, η belongs to the reduced state graph M'.

Proof By induction on the length of the prefixes η_i of η. The base case is that η_0 is a single node, which is an initial state in S. According to the reduction algorithms, all the initial states are included in S' as well. For the inductive step, assume that η_i is in M'. Then notice that η_{i+1} is obtained from η_i by appending a transition from $ample(last(\eta_i))$. □

The following three lemmas will be used to show that the path η that is constructed as the limit of the finite paths η_i contains all of the visible transitions of σ, and in the same order.

LEMMA 28 Let α be the first transition on θ_i. Then there exists $j > i$ such that α is the last transition of η_j, and for $i \leq k < j$, α is the first transition of θ_k.

Proof According to the above construction, if α is the first transition of θ_k, then either it is the first transition of θ_{k+1} (case **B**), or it will become the last transition of η_{k+1} (case **A**). We need to show that the first case cannot hold for every $k \geq i$. Suppose, on the contrary, that this is the case. Let $s_k = first(\theta_k)$. Consider the infinite sequence $s_i, s_{i+1}, \ldots$. According to the above construction, $s_{k+1} = \gamma_k(s_k)$ for some $\gamma_k \in ample(s_k)$. Moreover, because α is the first transition of θ_k and was not selected in case **A** to be moved to η_{k+1}, α must be in $enabled(s_k) \setminus ample(s_k)$. Because the number of states in S is finite, there is some state s_k that is the first to repeat on the sequence $s_i, s_{i+1}, \ldots$. Thus, there is a cycle $s_k, s_{k+1}, \ldots s_r$, with $s_r = s_k$, where α does not appear in any of the ample sets. This violates Condition **C3**. □

LEMMA 29 Let γ be the first visible transition on θ_i and $prefix_\gamma(\theta_i)$ be the maximal prefix of $tr(\theta_i)$ that does not contain γ. Then one of the following holds:

- γ is the first transition of θ_i and the last transition of η_{i+1}, or
- γ is the first visible transition of θ_{i+1}, the last transition of η_{i+1} is invisible, and $prefix_\gamma(\theta_{i+1}) \sqsubseteq prefix_\gamma(\theta_i)$.

Proof The first case of the lemma holds when γ is selected from $ample(s_i)$ and becomes the last transition of η_{i+1}, according to case **A** of the construction. If this does not happen, there exists another transition β that is appended to η_i to form η_{i+1}. The transition β cannot be visible. Otherwise, according to Condition **C2**, $ample(s_i) = enabled(s_i)$. By case **B1** of the construction, β must be the first transition of θ_i. But then β is a visible transition that precedes γ in θ_i, a contradiction.

There are three possibilities:

1. β appears on θ_i before γ (case **B1** in the construction),
2. β appears on θ_i after γ (case **B1** in the construction), or
3. β is independent of all the transitions of θ_i (case **B2** in the construction).

According to the above construction, in (1), $prefix_\gamma(\theta_{i+1}) \sqsubset prefix_\gamma(\theta_i)$ since β is removed from the prefix of θ_i before γ when constructing θ_{i+1}. In (2) and (3), $prefix_\gamma(\theta_{i+1}) = prefix_\gamma(\theta_i)$ since the prefix of θ_{i+1} that precedes the transition γ has the same transitions as the corresponding prefix of θ_i. □

LEMMA 30 Let v be a prefix of $vis(tr(\sigma))$. Then there exists a path η_i such that $v = vis(tr(\eta_i))$.

Proof By induction on the length of v. The base holds trivially for $|v| = 0$. In the inductive step we must prove that if $v\gamma$ is a prefix of $vis(tr(\sigma))$ and there is a path η_i such that $vis(tr(\eta_i)) = v$, then there is a path η_j with $j > i$ such that $vis(tr(\eta_j)) = v\gamma$. Thus, we need to show that γ will be eventually added to η_j for some $j > i$, and that no other visible transition will be added to η_k for $i < k < j$. According to case **A** in the construction, we may add a visible transition to the end of η_k to form η_{k+1} only if it appears as the first transition of θ_k. Lemma 29 shows that γ remains the first visible transition in successive paths θ_k after θ_i unless it is being added to some η_j. Moreover, the sequence of transitions before γ can only shrink. Lemma 28 shows that the first transition in each θ_k is eventually removed and added to the end of some η_l for $l > k$. Thus, γ as well is eventually added to some sequence η_j. □

THEOREM 12 The structures M and M' are stuttering equivalent.

Proof Each infinite path of M' that begins from an initial state must also be a path of M, for it is constructed by repeatedly applying transitions from the initial state. We need to show that for each path $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ in M, where s_0 is an initial state, there exists a path $\eta = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ in M' such that $\sigma \sim_{st} \eta$. We will show that the path η that is constructed above for σ is indeed stuttering equivalent to σ.

First, we show that σ and η have the same sequence of visible transitions, that is, $vis(tr(\sigma)) = vis(tr(\eta))$. According to Lemma 30, η contains the visible transitions of σ in the same order, because for any prefix of σ with m visible transitions, there is a prefix η_i of η with the same m visible transitions. On the other hand, σ must contain the visible transitions of η in the same order. Take any prefix η_i of η. According to Lemma 26, $\pi_i = \eta_i \circ \theta_i$ has the same visible transitions as $\pi_0 = \sigma$. Thus, σ has a prefix with the same sequence of visible transitions as η_i.

We construct two infinite sequences of indexes $0 = i_0 < i_1 < \ldots$ and $0 = j_0 < j_1 < \ldots$ that define corresponding stuttering blocks of σ and η, as required in the definition of stuttering. Assume that both $\sigma = \pi_0$ and η have at least n visible transitions. Let i_n be the length of the smallest prefix ξ_{i_n} of σ that contains exactly n visible transitions. Let j_n be the length of the smallest prefix η_{j_n} of η that contains the same sequence of visible transitions as ξ_{i_n}. Recall that η_{j_n} is a prefix of π_{j_n}. Then by part 3 of lemma 26, $L(s_{i_n}) = L(r_{j_n})$. By the definition of visible transitions we also know that if $n > 0$, for $i_{n-1} \leq k < i_n - 1$, $L(s_k) = L(s_{i_{n-1}})$. This is because i_{n-1} is the length of the smallest prefix $\xi_{i_{n-1}}$ of σ that contains exactly $n - 1$ visible transitions. Thus, there is no visible transition between i_{n-1} and $i_n - 1$. Similarly, for $j_{n-1} \leq l < j_n - 1$, $L(r_l) = L(r_{j_{n-1}})$.

If both σ and η have infinitely many visible transitions, then this process will construct two infinite sequences of indexes. In the case where σ and η contain only a finite number of visible transitions m, we have that for $k > i_m$, $L(s_k) = L(s_{i_m})$ and for $l > j_m$, $L(r_l) = L(r_{j_m})$. We then set for $k \geq m$, $i_{k+1} = i_k + 1$ and $j_{k+1} = j_k + 1$. By the above, for $k \geq 0$, the blocks of states $s_{i_k}, s_{i_k+1}, \ldots, s_{i_{k+1}-1}$ and $r_{j_k}, r_{j_k+1}, \ldots, r_{j_{k+1}-1}$ are corresponding stuttering blocks that have the same labeling. Thus, $\sigma \sim_{st} \eta$. □

10.7 Partial Order Reduction in SPIN

SPIN [138, 140] is an on-the-fly LTL model checker that uses explicit state enumeration and the partial order reduction. It was developed at Bell Laboratories by Gerard Holzmann and Doron Peled. The tool is used primarily for verifying asynchronous software systems, in particular communication protocols. It can check a model of a program for deadlocks or unreachable code or determine if it satisfies an LTL specification, based on the translation algorithm [124] described in Section 9.4. The tool uses the partial order reduction [139, 209] to limit the state space that is searched.

The input language for SPIN, called PROMELA, was developed by Gerard Holzmann. This language uses syntactic constructs from several different programming languages. PROMELA expressions are inherited from the language C [154]. Thus, the language has the operators '==' (equals), '!=' (not equals), '||' (logical or), '&&' (logical and), and '%' (reminder modulo an integer). Assignment is denoted by a single '=' symbol. Negation is denoted by prefixing a boolean expression by the operator '!'.

The syntax for communication commands is inherited from CSP [137]. Sending a message that contains the tag `tg` and the values $val_1, val_2, \ldots, val_n$ over channel `ch` is denoted by

$$\texttt{ch!tg}(val_1, val_2, \ldots, val_n)$$

in the sending process. Receiving a message with tag `tg` over channel `ch` is denoted by

$$\texttt{ch?tg}(var_1, var_2, \ldots, var_n)$$

in the receiving process. The message consists of n values that are stored in the variables $var_1, var_2, \ldots, var_n$. SPIN also allows untagged message passing. The language implements both message passing with queues and message passing using handshaking. In message passing with queues, a channel of some fixed length temporarily stores the values sent, so that the sending process can proceed to its next command, even if the receiving process is not yet ready to process the incoming data. In message passing with handshaking, a channel is defined in SPIN to be of length 0. Then, a send and a receive command

if	**do**
:: $guard_1 -> S_1$	:: $guard_1 -> S_1$
:: $guard_2 -> S_2$	:: $guard_2 -> S_2$
⋮	⋮
:: $guard_n -> S_n$	:: $guard_n -> S_n$
fi	**od**

Figure 10.10
Conditionals and loops in SPIN.

with the same channel and tag (if a tag is present) are executed simultaneously. This results in the assignment of val_i to var_i, for $1 \leq i \leq n$.

The conditional constructs and loops are based on Dijkstra's *Guarded Commands* [95] and use the syntax in Figure 10.10.

Each guard consists of a condition, a communication command, or both. In order for a guard to be *passable*, its condition must hold, and its communication command must not be blocked. In message passing with queues, a send command is blocked when the queue is full, and a receive command is blocked when the queue is empty. In message passing based on handshaking, communication is blocked when only one of the communicating processes is ready to send or to receive.

When executing the `if` construct and at each iteration of the `do` loop, one of the passable guards $guard_i$ is selected nondeterministically and then the corresponding command S_i is executed. A `do` loop repeats until either a *goto* command forces a branch to a particular label outside its scope, or a `break` command forces a skip to the first command after the `do` loop.

The reduction obtained by using the ample set technique described in Section 10.3 is demonstrated using the *leader election* algorithm developed by Dolev, Klawe, and Rodeh [102]. This algorithm operates on a ring of N processes. Each process initially has a unique number. The purpose of this algorithm is to find the largest number assigned to a process. The ring of processes is unidirectional; hence, each process can receive messages from its left and send messages to its right.

Initially, each process P_i is *active* and holds some integer value in its local variable `my_val`. As long as P_i is active, it is *responsible* for some value. This value may change during the execution of the algorithm. The current value of P_i is held in the variable `max`. A process becomes *passive* when it finds out that it does not hold a value that can be the maximum one. A passive process can only pass messages from left to right. Each active process P_i sends its own value to the right and then waits to receive the value of the closest active

process P_j on its left. This value is received using a communication command tagged with `one`.

If the value received by P_i is the same as the value it sent, then P_i can conclude that it is the only active process and, hence, its value is the maximum. Then process P_i sends this value to the right with the tag `winner`. Every other process receives this value and sends it to the right exactly once, so that all the processes can learn the winning number.

If the value received by P_i is not the same as the value it sent, then P_i waits for a second message, tagged with `two`, that includes the value of the second closest active process on its left P_k. Then, P_i compares its own value with the two values it received from P_j and P_k. If the value received from P_j is the greatest among the three, then P_i keeps this value. That is, P_i becomes responsible for the role of the closest active process P_j. Otherwise, P_i becomes passive.

The execution of the algorithm can be divided into phases. In each *phase*, except the last, all of the active processes receive messages tagged with `one` and `two`. In the last phase, the surviving process receives its own value via a message tagged with `one` and then this value is propagated around the ring.

The protocol guarantees low message complexity $O(N \times log(N))$. This complexity bound holds because at least half of the active processes become passive in each phase. To see this, consider the case where P_i remains active. Then the value of P_j must be bigger than the values of P_i and P_k. If P_j also survives, then the value of P_k must be larger than the value of P_j. This is a contradiction. Thus, in each phase except for the last, if a process remains active, the first active process to its left must become passive. In each phase, the number of messages passed is limited to $2 \times N$, since each process receives two messages from its left neighbor.

The PROMELA code for the leader election algorithm appears in Figure 10.11. We omit the code for initializing the processes. This includes assigning a distinct number to each process and starting the execution of that process. The channel $q[(i+1)\%N]$ is used to send messages from process P_i to process $P_{i+1\%N}$, where $\%N$ denotes the reminder modulo N.

The property that we checked is given by the LTL formula

$$noLeader \ \mathbf{U} \ \mathbf{G} \ oneLeader.$$

This formula asserts that in each execution there is no leader until some time in the future when a leader is selected. From that point onward, there is exactly one leader. The predicates *noLeader* and *oneLeader* are defined as `number_leaders == 0` and `number_leaders == 1`, respectively.

```
#define noLeader        (number_leaders == 0)
#define oneLeader       (number_leaders == 1)

byte number_leaders = 0;

#define N       6       /* number of processes in the ring */
#define L       12       /* 2xN */
byte I;

mtype = { one, two, winner };
chan q[N] = [L] of { mtype, byte};

proctype P (chan in, out; byte my_val)
{       bit Active = 1, know_winner = 0;
        byte number, max = my_val, neighbor;

        out!one(my_val);
        do
        :: in?one(number) ->  /*Get left active neighbor value*/
           if
           :: Active ->
              if
              :: number != max ->
                 out!two(number); neighbor = number
              :: else ->
                 know_winner = 1; out!winner(number);
              fi
           :: else ->
              out!one(number)
           fi
```

Figure 10.11
The leader election protocol in PROMELA.

The negation of the checked property is automatically translated into a Büchi automaton, based on the algorithm described in Section 9.4. An additional minimization stage combines nodes with the same branching structure. The automaton is described using a special syntactical construct of PROMELA called the *never claim*. The reason for this name is that the automaton, obtained by translating the negation of the checked property, repre-

```
:: in?two(number) ->  /*Get second left active neighbor value*/
   if
   :: Active ->
      if
      :: neighbor > number && neighbor > max ->
         max = neighbor; out!one(neighbor)
      :: else ->
         Active = 0 /* Becomes passive */
      fi
   :: else ->
      out!two(number)
   fi

:: in?winner(number) ->
   if
   :: know_winner
   :: else -> out!winner(number)
   fi;
   break
od
```

Figure 10.11 (continued)

sents the computations that should never happen. The never claim for the above property is shown in Figure 10.12. The label of each initial node contains the word `init` and the label of each accepting node contains the word `accept`.

SPIN intersects the automaton extracted from the program and the never claim automaton. This intersection is done on-the-fly, using the double-DFS algorithm presented in Section 9.3 and the partial order reduction. If the intersection is not empty, an error trace is reported.

The experimental results are summarized in the table in Figure 10.13. The experiments were conducted on an SGI *Challenge* machine. The memory in the table is given in megabytes. Verifying the algorithm with five and six processes without using the partial order reduction did not terminate. The table indicates that the case of five processes without partial order reduction was still running after forty hours. The results of this experiment clearly demonstrates how the partial order reduction is able to alleviate the state explosion problem.

```
never {    /* !(noLeader U [] oneLeader) */
T0_init:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto accept_S1
    fi;
accept_S1:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto T0_init
    fi;
accept_S9:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto T0_init
    fi;
accept_S28:
    if
    :: (1) -> goto T0_S28
    :: (! ((oneLeader))) -> goto accept_all
    fi;
T0_S9:
    if
    :: (! ((noLeader))) -> goto T0_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_S28
    :: (! ((noLeader)) && ! ((oneLeader))) -> goto accept_all
    :: (! ((oneLeader))) -> goto accept_S9
    :: (1) -> goto T0_S9
    :: (! ((oneLeader))) -> goto accept_S1
    fi;
T0_S28:
    if
    :: (1) -> goto T0_S28
    :: (! ((oneLeader))) -> goto accept_all
    fi;
accept_all:
    skip
}
```

Figure 10.12
The never claim for the specification.

	Non-reduced			Reduced		
Procs:	States	Memory	Time	States	Memory	Time
3	15929	1.801	13.8 sec	1435	1.493	0.6 sec
4	522255	15.727	9.3 min	8475	1.698	3.5 sec
5		>128	>40 hours	57555	3.234	28.7 sec
6				434083	15.625	4.1 min

Figure 10.13
Experimental results for the partial order reduction.

11 Equivalences and Preorders between Structures

In this chapter we will show how to avoid the state explosion problem by developing techniques that replace a large structure by a smaller structure which satisfies the same properties. We have already seen one example of this technique in Chapter 10 where the partial order reduction was used to reduce the size of structures while preserving the truth of LTL formulas that do not involve the next-time operator. More generally, given a logic $\mathcal{L}$ and a structure M, we would like to find a smaller structure M' that satisfies exactly the same set of formulas of the logic $\mathcal{L}$ as M. In order to accomplish this goal, we need a notion of equivalence between structures that can be efficiently computed and guarantees that two structures satisfy the same set of formulas in $\mathcal{L}$. We first consider the logic CTL* and *bisimulation equivalence* [207].

It is convenient to include a set of initial states S_0 and a set of atomic propositions AP with every structure M. Thus, a typical structure is $M = (AP, S, R, S_0, L)$. If fairness is also considered, then $M = (AP, S, R, S_0, L, F)$. Sometimes it is necessary to transform a structure that does not have fairness assumptions into one that does, while preserving the set of paths considered as computations. This can be accomplished by letting $F = \{S\}$.

Let $M = (AP, S, R, S_0, L)$ and $M' = (AP, S', R', S'_0, L')$ be two structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a *bisimulation relation* between M and M' if and only if for all s and s', if $B(s, s')$ then the following conditions hold:

1. $L(s) = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$ there is s'_1 such that $R'(s', s'_1)$ and $B(s_1, s'_1)$.
3. For every state s'_1 such that $R'(s', s'_1)$ there is s_1 such that $R(s, s_1)$ and $B(s_1, s'_1)$.

The structures M and M' are *bisimulation equivalent* (denoted $M \equiv M'$) if there exists a bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0, s'_0)$.

Figures 11.1 and 11.2 demonstrate simple examples of bisimulation equivalent structures. The figures show that unwinding a structure or duplicating some part of a structure may result in a bisimulation equivalent structure. Figure 11.3, on the other hand, shows two structures that are not bisimulation equivalent. In order to see this, note that the state labeled with b in M' does not correspond to any of the states labeled with b in M because none of these states have both a successor labeled by c and a successor labeled by d.

The following lemma is important in establishing the connection between CTL* and bisimulation equivalence. We say that two paths $\pi = s_0 s_1, \ldots$ in M and $\pi' = s'_0 s'_1, \ldots$ in M' *correspond* if and only if for every $i \geq 0$, $B(s_i, s'_i)$.

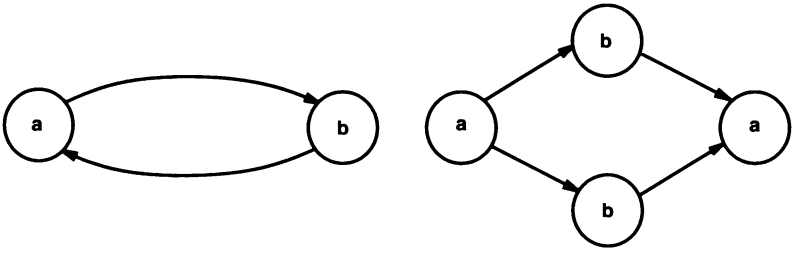


Figure 11.1
Unwinding preserves bisimulation.

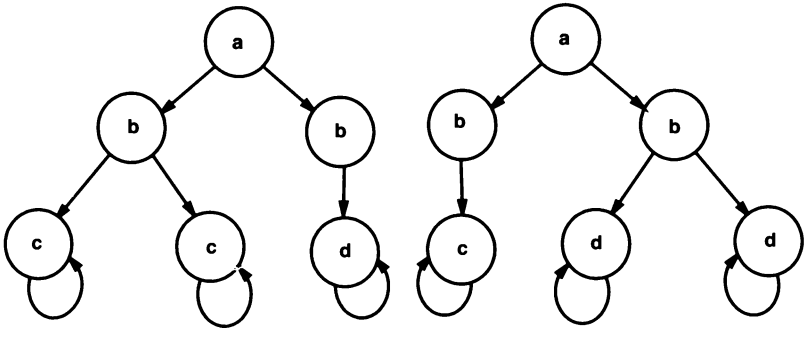


Figure 11.2
Duplication preserves bisimulation.

LEMMA 31 Let s and s' be two states such that $B(s, s')$. Then for every path starting from s there is a corresponding path starting from s', and for every path starting from s' there is a corresponding path starting from s.

Proof Let $B(s, s')$ and let $\pi = s_0 s_1, \ldots$ be a path from $s = s_0$. We construct a corresponding path $\pi' = s'_0 s'_1, \ldots$ from $s' = s'_0$ by induction. It is clear that $B(s_0, s'_0)$. Assume $B(s_i, s'_i)$ for some i. We will show how to choose s'_{i+1}. Because $B(s_i, s'_i)$ and $R(s_i, s_{i+1})$, there must be a successor t' of s'_i such that $B(s_{i+1}, t')$. We choose s'_{i+1} to be t'.

Given a path π' from s', the construction of a path π from s is similar. □

The next lemma shows that if two states are bisimilar, then they satisfy the same set of CTL* state formulas. Furthermore, if two paths correspond, then they satisfy the same set of path formulas.

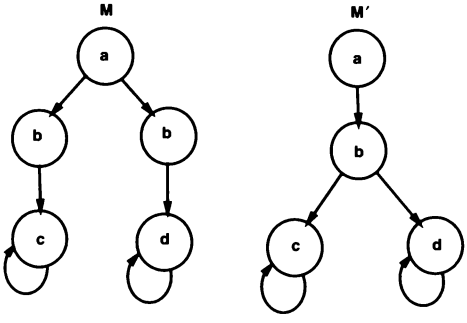


Figure 11.3
Two nonbisimilar structures.

LEMMA 32 Let f be either a state formula or a path formula. Assume that s and s' are bisimilar states and that π and π' are corresponding paths. Then,

- if f is a state formula, then $s \models f \Leftrightarrow s' \models f$, and
- if f is a path formula, then $\pi \models f \Leftrightarrow \pi' \models f$.

Proof We prove the lemma by induction on the structure of f.
Base: $f = p$ for $p \in AP$. Because $B(s, s')$, we know that $L(s) = L'(s')$. Thus, $s \models p$ if and only if $s' \models p$.
Induction: Consider the following cases.

1. $f = \neg f_1$, a state formula.

$$\begin{aligned} s \models f &\Leftrightarrow s \not\models f_1 \\ &\Leftrightarrow s' \not\models f_1 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow s' \models f \end{aligned}$$

The same reasoning holds if f is a path formula.

2. $f = f_1 \vee f_2$, a state formula.

$$\begin{aligned} s \models f &\Leftrightarrow s \models f_1 \quad \text{or} \quad s \models f_2 \\ &\Leftrightarrow s' \models f_1 \quad \text{or} \quad s' \models f_2 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow s' \models f \end{aligned}$$

We can also use this argument if f is a path formula.

3. $f = f_1 \wedge f_2$, a state formula. This case is similar to the previous case. Furthermore, the same argument can be used if f is a path formula.

4. $f = \mathbf{E}\, f_1$, a state formula. Suppose that $s \models f$. Then there is a path π_1 starting from s such that $\pi_1 \models f_1$. By Lemma 31, there is a corresponding path π_1' in M' starting from s'. So by the induction hypothesis, $\pi_1 \models f_1$ if and only if $\pi_1' \models f_1$. Therefore, $s' \models \mathbf{E}\, f_1$. The same argument can be used to prove that if $s' \models f$ then $s \models f$.

5. $f = \mathbf{A}\, f_1$, a state formula. The argument for this case is similar to the argument for $f = \mathbf{E}\, f_1$.

6. $f = f_1$, where f is a path formula and f_1 is a state formula. Although the lengths of f and f_1 are the same, we can imagine that $f = \mathbf{path}(f_1)$, where **path** is a special operator which converts a state formula into a path formula. Therefore, we are simplifying f by dropping this **path** operator. If s_0 and s_0' are the first states of π and π', respectively, then

$$\begin{aligned} \pi \models f &\Leftrightarrow s_0 \models f_1 \\ &\Leftrightarrow s_0' \models f_1 \qquad \text{(induction hypothesis)} \\ &\Leftrightarrow \pi' \models f \end{aligned}$$

7. $f = \mathbf{X}\, f_1$, a path formula. Suppose $\pi \models f$. By the definition of the next time operator, $\pi^1 \models f_1$. Because π and π' correspond, so do π^1 and π'^1. Therefore, by the induction hypothesis, $\pi'^1 \models f_1$, and so $\pi' \models f$. The same argument can be used to prove that if $\pi' \models f$ then $\pi \models f$.

8. $f = f_1\, \mathbf{U}\, f_2$, a path formula. Suppose that $\pi \models f_1\, \mathbf{U}\, f_2$. By the definition of the until operator, there is a k such that $\pi^k \models f_2$ and for all $0 \leq j < k$, $\pi^j \models f_1$. Because π and π' correspond, so do π^j and π'^j for any j. Therefore, by the induction hypothesis, $\pi'^k \models f_2$ and for all $0 \leq j < k$, $\pi'^j \models f_1$. Therefore, $\pi' \models f$. The same argument can be used to prove that if $\pi' \models f$ then $\pi \models f$.

9. $f = f_1\, \mathbf{R}\, f_2$, a path formula. The argument in this case is similar to the argument for $f = f_1\, \mathbf{U}\, f_2$. □

The next theorem is a consequence of the preceding lemma.

THEOREM 13 If $B(s, s')$ then for every CTL* formula f, $s \models f \Leftrightarrow s' \models f$.

If two structures are bisimulation equivalent, then every initial state of one is bisimilar to some initial state of the other. Because a structure satisfies a formula if and only if each of its initial states satisfies the formula, both structures will satisfy the same set of CTL* formulas.

THEOREM 14 If $M \equiv M'$ then for every CTL* formula f, $M \models f \Leftrightarrow M' \models f$.

The converse of this theorem is also true. If two structures satisfy the same set of CTL* formulas then they are bisimulation equivalent. In fact, we can show that if two structures satisfy the same CTL formulas they are bisimulation equivalent. It follows that if two structures can be *distinguished* by a formula of CTL* (i.e., there is a CTL* formula that is true of one structure and not of the other) then they can also be distinguished by a formula of CTL. These results are described in [32]. Note that this result does not imply that CTL* and CTL have the same expressive power. For comparing the expressiveness of two logics, we view a formula as defining the set of models where the formula is true. For CTL to have the same expressiveness as CTL*, it would be necessary for every formula of CTL* to have a corresponding formula of CTL that defines the same set of models. This, however, is known not to be true [105]. Instead, the previous result implies that for every model, there exists a CTL formula that is true in that model but not in any other, inequivalent model.

The notion of bisimulation equivalence can be extended to structures with *fairness constraints*. Let M and M' be two structures with fairness constraints. Assume that both have the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a *fair bisimulation relation* between M and M' if and only if for all s and s', if $B(s, s')$ the following conditions hold:

1. $L(s) = L'(s')$.
2. For every *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M there is a *fair* path $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' such that for all $i \geq 0$, $B(s_i, s'_i)$.
3. For every *fair* path $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' there is a *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M such that for all $i \geq 0$, $B(s_i, s'_i)$.

In this case, two structures M and M' are *fair bisimulation equivalent* (denoted $M \equiv_F M'$) if there exists a fair bisimulation relation B such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $B(s_0, s'_0)$. In addition, for every initial state $s'_0 \in S'_0$ in M' there is an initial state $s_0 \in S_0$ in M such that $B(s_0, s'_0)$. If the semantics of CTL* is given with respect to fair paths then we can prove an analog of Theorem 14 for fair structures.

THEOREM 15 If $M \equiv_F M'$, then for every CTL* formula f interpreted over fair paths, $M \models_F f \Leftrightarrow M' \models_F f$.

The proof of this theorem is similar to the proof of the previous theorem and is omitted.

Sometimes bisimulation equivalence does not result in a significant reduction in the number of states. By restricting the logic and relaxing the requirement that the structures

should satisfy exactly the same formulas, a greater reduction can be obtained. In order to achieve this goal we introduce the notion of a *simulation relation*. Simulation is closely related to bisimulation. Bisimulation guarantees that two structures have the same behaviors. Simulation, on the other hand, relates a structure to an *abstraction* of the structure. Because the abstraction can hide some of the details of the original structure, it might have a smaller set of atomic propositions. The simulation guarantees that every behavior of a structure is also a behavior of its abstraction. However, the abstraction might have behaviors that are not possible in the original structure. For example, in an actual implementation some event always occurs within twenty execution steps. But in an abstraction, this event may occur after any number of execution steps.

Given two structures M and M' with $AP \supseteq AP'$, a relation $H \subseteq S \times S'$ is a *simulation relation* [195] between M and M' if and only if for all s and s', if $H(s, s')$ then the following conditions hold.

1. $L(s) \cap AP' = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$, there is a state s'_1 with the property that $R'(s', s'_1)$ and $H(s_1, s'_1)$.

We say that M' *simulates* M (denoted by $M \preceq M'$) if there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s'_0 in M' for which $H(s_0, s'_0)$.

Consider again the structures in Figure 11.3. We saw previously that they were not bisimulation equivalent. We will now show that the structure M is smaller in the simulation preorder than the structure M'. We choose a simulation relation $\preceq$ that associates with each state in M the state in M' that has the same label. The relation $\preceq$ has the property that if it associates a state s with a state s', then every successor of s has a corresponding successor of s'. On the other hand, M does not simulate M', because the state in M' labeled with b does not have a corresponding state in M.

Now, we show that simulation is a preorder, that is, a reflexive and transitive relation.

LEMMA 33 $\preceq$ is a preorder on the set of structures.

Proof The relation $H = \{ (s, s) \mid s \in S \}$ is a simulation between M and M, so $\preceq$ is reflexive. Thus it only remains to show that $\preceq$ is transitive. Assume $M \preceq M'$ and $M' \preceq M''$. Let H_0 be a simulation between M and M', and let H_1 be a simulation between M' and M''. Define H_2 as the relational product of H_0 and H_1, that is,

$$H_2 = \{ (s, s'') \mid \exists s' \, [H_0(s, s') \wedge H_1(s', s'')] \}.$$

If $s_0 \in S_0$, then by the definition of simulation, there exists $s'_0 \in S'_0$ such that $H_0(s_0, s'_0)$. Similarly, there exists $s''_0 \in S''_0$ such that $H_1(s'_0, s''_0)$, and hence $H_2(s_0, s''_0)$.

Suppose $H_2(s, s'')$, and let s' be such that $H_0(s, s')$ and $H_1(s', s'')$. By the definition of simulation, $L(s) \cap AP' = L'(s')$ and $L'(s') \cap AP'' = L''(s'')$. Then because $AP' \supseteq AP''$, we have $L(s) \cap AP'' = L''(s'')$. Let $R(s, s_1)$ be a transition in M from s. Then there exists a transition $R(s', s_1')$ in M' such that $H_0(s_1, s_1')$. Because H_1 is a simulation, there exists a transition $R''(s'', s_1'')$ in M'' such that $H_1(s_1', s_1'')$. Hence $H_2(s_1, s_1'')$, and H_2 is a simulation between M and M''. Thus $M \preceq M''$. □

The following lemma is the analog of lemma 31 for simulation relations. In this case, we also say that paths $\pi = s_0 s_1 \ldots$ in M and $\pi' = s_0' s_1' \ldots$ in M' *correspond* if and only if for every i, $H(s_i, s_i')$.

LEMMA 34 Assume that s and s' are states such that $H(s, s')$. Then for every path π starting from s there is a corresponding path π' starting from s'.

THEOREM 16 Suppose $M \preceq M'$. Then for every ACTL* formula f (with atomic propositions in AP'), $M' \models f$ implies $M \models f$.

Intuitively, this theorem is true because formulas in ACTL* describe properties that are quantified over all possible behaviors of a structure. Because every behavior of M is a behavior of M', every formula of ACTL* that is true in M' must also be true in M. A formal proof can be obtained from Lemma 34 by using an argument similar to the one used to establish Theorem 14. This theorem is very useful for model checking when M is much more complicated than M'. If it is possible to establish an ACTL* property f for M', then f will also be true of the more complex model M. On the other hand, if f does not hold for M', then f may or may not hold for M. Thus, if a counterexample is obtained when checking f on M', it still necessary to check whether the counterexample actually corresponds to an error in M. This theorem will be used frequently in subsequent chapters.

Figure 11.4 illustrates the difference between simulation and bisimulation. The two structures in the figure are not bisimulation equivalent, but each simulates the other. In order to show that M simulates M' we choose a simulation relation that associates both states 3 and 4 in M' with the state 1 in M. Each of the other states in M' is associated with all of the states in M that have the same label.

To see that M' simulates M, we choose the relation that associates both states 1 and 2 in M with state 3 in M'. All of the other states of M are associated with states in M' as in the previous case.

M and M' are not bisimulation equivalent since no state in M can be associated with state 4 in M'. Another way to see why this is true is to use Theorem 13, which states that two bisimulation equivalent structures satisfy the same CTL formulas. It is easy to see that the CTL formula $\mathbf{AG}(b \rightarrow \mathbf{EX}\, c)$ it true in M but false in M'. However, because

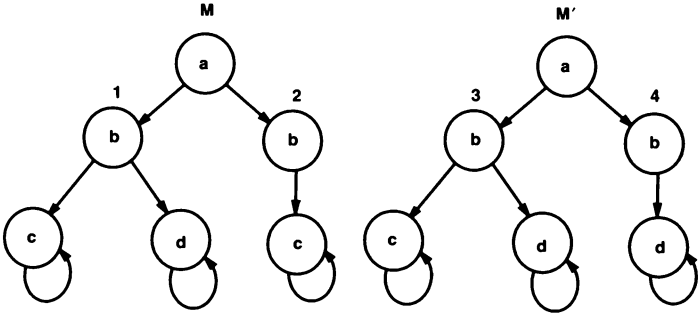


Figure 11.4
Simulation equivalent structures that are not bisimilar.

of Theorem 16, the two structures do satisfy the same ACTL formulas. This shows that equivalence with respect to ACTL is different from equivalence with respect to CTL.

Simulation can be extended to fair structures in the same way that bisimulation is extended to fair structures. Let M and M' be two structures with fairness constraints. Assume that $AP \supseteq AP'$. The relation $H \subseteq S \times S'$ is a *fair simulation relation* between M and M' if and only if for all s and s', if $H(s, s')$ then the following conditions hold:

1. $L(s) \cap AP' = L'(s')$.
2. For every *fair* path $\pi = s_0 s_1 \ldots$ from $s = s_0$ in M, there is a fair *path* $\pi' = s'_0 s'_1 \ldots$ from $s' = s'_0$ in M' such that for all $i \geq 0$, $H(s_i, s'_i)$.

We write $M \preceq_F M'$ if there exists a fair simulation relation H such that for every initial state $s_0 \in S_0$ in M there is an initial state $s'_0 \in S'_0$ in M' such that $H(s_0, s'_0)$. It is easy to show that $\preceq_F$ determines a preorder on fair structures. When it is clear from the context that we are dealing with fair simulation, we will sometimes use $\preceq$.

Every fair behavior of M is a fair behavior of M'. Thus, if the semantics of ACTL* is given with respect to fair paths then we can prove the following theorem.

THEOREM 17 If $M \preceq_F M'$, then for every ACTL* formula f interpreted over fair paths, $M' \models_F f$ implies $M \models_F f$.

11.1 Equivalence and Preorder Algorithms

We next consider algorithms that determine whether two structures are bisimulation equivalent or whether one structure precedes another in the simulation preorder. Bisimulation

equivalence is easy to check if both structures are *deterministic*, that is, each has a single initial state, and if $R(s,t)$ and $R(s,u)$, then $L(t) \neq L(u)$. The *language of a structure* is the set of sequences of labelings that occur along all paths that start from initial states. It can be shown that two deterministic structures are bisimulation equivalent if and only if they have the same language. Efficient algorithms are known for checking language equivalence for deterministic structures [60]. These algorithms can be used to check bisimulation equivalence for deterministic structures.

We now present a general algorithm that handles both deterministic and nondeterministic structures that do not include fairness constraints. Let M and M' be two structures with the same set of atomic propositions AP. We define a sequence of relations $B_0^*, B_1^*, \ldots$ on $S \times S'$ as follows:

1. $B_0^*(s,s')$ if and only if $L(s) = L'(s')$.
2. $B_{n+1}^*(s,s')$ if and only if

- $B_n^*(s,s')$, and
- $\forall s_1[R(s,s_1) \implies \exists s_1'[R'(s',s_1') \wedge B_n^*(s_1,s_1')]]$, and
- $\forall s_1'[R'(s',s_1') \implies \exists s_1[R(s,s_1) \wedge B_n^*(s_1,s_1')]]$.

We write $B^*(s,s')$ if and only if $B_i^*(s,s')$ for all $i \geq 0$. Note that, by definition $B_i^* \supseteq B_{i+1}^*$ for all $i \geq 0$. Thus, because M and M' are finite, there is an n such that $B_n^* = B_{n+1}^*$. It is easy to see that B_n^* is exactly B^*.

Two structures M and M' are *B^*-equivalent* if for every initial state $s_0 \in S_0$ in M there is an initial state $s_0' \in S_0'$ in M' such that $B^*(s_0,s_0')$. In addition, for every initial state $s_0' \in S_0'$ in M' there is an initial state $s_0 \in S_0$ in M such that $B^*(s_0,s_0')$. It is easy to see that B^* is a bisimulation between M and M'. In fact, we will show that B^* is the *largest* such bisimulation, that is, every bisimulation between M and M' is included in B^*. (Inclusion between bisimulation relations is interpreted as set inclusion.) Thus, M and M' are bisimulation equivalent if and only if they are B^*-equivalent.

LEMMA 35 B^* is the largest bisimulation between M and M' (in terms of set inclusion).

Proof It is sufficient to prove that if B is a bisimulation between M and M', then B is contained in B_i^* for every $i \geq 0$. We show this by induction on i. Clearly, B is contained in B_0^*, since any pair of states in B have the same labeling. Assume that B is contained in B_n^* and that $B(s,s')$. Let $R(s,s_1)$ be a transition in M. Because B is a bisimulation, there exists a state s_1' such that $R'(s',s_1')$ is a transition in M' and $B(s_1,s_1')$. Since B is contained in B_n^*, we have that $B_n^*(s_1,s_1')$. The third requirement can be proved in a similar manner. Thus, $B_{n+1}^*(s,s')$. □

As explained above, the finiteness of the structures guarantees that there exists some n such that $B^* = B^*_n$. Thus, the definition gives an algorithm for computing the largest bisimulation between two structures. If an explicit state representation is used for the transition relations, then the algorithm has polynomial time complexity in the size of the two structures. A more efficient polynomial algorithm for this case is given in [206]. If OBDDs are used to represent the transition relations, then the definition can be used directly to compute the largest bisimulation—it just describes the computation of the greatest fixpoint of an appropriate function.

Algorithms for checking fair bisimulation have not been widely investigated. If the structures are deterministic, then an efficient algorithm, based on language equivalence, can also be given in this case. The only change that is necessary is to restrict the language of a structure to fair paths. With this change it is possible to prove that two structures are fair bisimulation equivalent if and only if they are language equivalent with respect to fair paths. Thus, algorithms that check language equivalence for fair structures [60] can be used to handle this case. A general procedure that also handles nondeterministic structures is given in [13]. This problem is PSPACE-complete in the size of the structures [159].

Each of the algorithms mentioned above can be adapted to check the simulation preorder between two structures M and M'. Language inclusion replaces language equivalence in the deterministic case. For the general case without fairness, we define a sequence of relations $H^*_0, H^*_1, \ldots$ on $S \times S'$ as follows:

1. $H^*_0(s, s')$ if and only if $L(s) \cap AP' = L'(s')$;
2. $H^*_{n+1}(s, s')$ if and only if
- $H^*_n(s, s')$, and
- $\forall s_1[R(s, s_1) \implies \exists s'_1[R'(s', s'_1) \wedge H^*_n(s_1, s'_1)]]$;

The procedure is guaranteed to terminate because the structures are finite. We write $H^*(s, s')$ if and only if $H^*_i(s, s')$ for all $i \geq 0$. As in the previous case, H^* is the largest simulation relation between the two structures M and M'. Thus, M' simulates M if and only if for every $s_0 \in S_0$ in M there is a state $s'_0 \in S'_0$ in M' such that $H^*(s_0, s'_0)$.

11.2 Tableau Construction

In this section, we give a tableau construction for ACTL formulas. Similar constructions for LTL were given in Chapter 6.7 and in Section 9. We show that the tableau $\mathcal{T}_f$ of an ACTL formula f is a *maximal model* for the formula under the relation $\preceq_F$. This is the key

property of the tableau construction. Because of this property, the tableau can be used as an assumption on the environment of a process in assume-guarantee reasoning (Chapter 12) by composing it with the process before model checking. Discharging the assumption is simply a matter of checking that the environment satisfies the formula. We also indicate how the tableau can be used to do temporal reasoning.

The tableau presented here differs somewhat from previous tableau constructions for LTL.

- For every structure M', the tableau is required to satisfy

$$M' \models_F f \text{ iff } M' \preceq_F \mathcal{T}_f.$$

In particular, we need $\mathcal{T}_f \models f$. When f contains eventualities (i.e., formulas of the form $\mathbf{A}[g\ \mathbf{U}\ h]$) this can be achieved by adding fairness constraints to the tableau and by considering only fair paths. Thus, $\mathcal{T}_f$ must be a fair Kripke structure.

- Because the structure M' under consideration is also a fair structure, it might contain states that do not have fair paths starting from them (note that this situation may occur even if the transition relation is total). Such states are characterized by the formula $\mathbf{AX}\ \mathit{False}$. This formula is added to the set of elementary formulas of f and will be included in every tableau state that simulates a state in M' that is not the beginning of a fair path. In addition to $\mathbf{AX}\ \mathit{False}$ we include the formulas *True* and *False* as subformulas of f.
- The nonpropositional elementary formulas in this construction are of the form $\mathbf{AX}\ g$ rather than $\mathbf{X}\ g$.

For the remainder of this section, fix an ACTL formula f. We now describe the construction of the tableau $\mathcal{T}_f$ for f in detail. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $\mathcal{T}_f = (AP_f, S_T, R_T, S_0^T, L_T, F_T)$. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively as follows:

1. $el(p) = el(\neg p) = \{p\}$ if $p \in AP_f$.
2. $el(g_1 \vee g_2) = el(g_1 \wedge g_2) = el(g_1) \cup el(g_2)$.
3. $el(\mathbf{AX}\ g_1) = \{\mathbf{AX}\ g_1\} \cup el(g_1)$.
4. $el(\mathbf{A}[g_1\ \mathbf{U}\ g_2]) = \{\mathbf{AX}\ \mathit{False}, \mathbf{AX}(\mathbf{A}[g_1\ \mathbf{U}\ g_2])\} \cup el(g_1) \cup el(g_2)$.
5. $el(\mathbf{A}[g_1\ \mathbf{R}\ g_2]) = \{\mathbf{AX}\ \mathit{False}, \mathbf{AX}(\mathbf{A}[g_1\ \mathbf{R}\ g_2])\} \cup el(g_1) \cup el(g_2)$.

The set of states S_T of the tableau is $\mathcal{P}(el(f))$. The labeling function L_T is defined so that each state is labeled by the set of atomic propositions contained in the state. In order to

specify the set of initial states and the transition relation R_T, we need an additional function *sat* that associates with each subformula g of f a set of states in S_T. Intuitively, $sat(g)$ will be the set of states that satisfy g.

1. $sat(True) = S_T$ and $sat(False) = \emptyset$.
2. $sat(g) = \{s \mid g \in s\}$ where $g \in el(f)$.
3. $sat(\neg g) = \{s \mid g \notin s\}$ where g is an atomic proposition. Recall that only atomic propositions can be negated in ACTL.
4. $sat(g \vee h) = sat(g) \cup sat(h)$.
5. $sat(g \wedge h) = sat(g) \cap sat(h)$.
6. $sat(\mathbf{A}[g\ \mathbf{U}\ h]) = \big(sat(h) \cup \big(sat(g) \cap sat(\mathbf{AX}(\mathbf{A}[g\ \mathbf{U}\ h]))\big)\big) \cup sat(\mathbf{AX}\ False)$.
7. $sat(\mathbf{A}[g\ \mathbf{R}\ h]) = \big(sat(h) \cap \big(sat(g) \cup sat(\mathbf{AX}(\mathbf{A}[g\ \mathbf{R}\ h]))\big)\big) \cup sat(\mathbf{AX}\ False)$.

The set of initial states of the tableau is $S_0^T = sat(f)$. We want the transition relation to have the property that each elementary formula in a state is true of that state. Clearly, if $\mathbf{AX}\ g$ is in some state s, then all the successors of s must satisfy g. On the other hand, if $\mathbf{AX}\ g$ is not in s, then s does not satisfy $\mathbf{AX}\ g$. Hence, s may have successors that satisfy g and others that do not. The definition for the transition relation R_T is

$$R_T(s_1, s_2) = \bigwedge_{\mathbf{AX}g \in el(f)} s_1 \in sat(\mathbf{AX}\ g) \Rightarrow s_2 \in sat(g).$$

Note that the definition of R_T differs from the one used in the LTL tableau construction in Section 6.7. There, if a state does not include $\mathbf{X}\ g$ then it satisfies $\neg\ \mathbf{X}\ g$, which is equivalent to $\mathbf{X}\ \neg g$. Thus, all of its successors must satisfy $\neg g$. This is accomplished by using $\Leftrightarrow$ in the definition of R_T for LTL.

We must also add an acceptance condition to guarantee that *eventuality* properties are fulfilled. The acceptance condition should restrict the set of (fair) paths so that:

- For every (fair) path π, for every elementary formula $\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h]$ of f, and for every state s on π, if $s \in sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$ then there is a later state t on π such that $t \in sat(h)$.

This can be enforced by a set of fairness constraints because of the following observation. Let s be a state in $sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$ and let state t be a successor of s under R_T; then either $t \in sat(h)$ or $t \in sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$. Thus, if $s \in sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$, then all succeeding states must also be in $sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$ unless we reach a state in $sat(h)$. The only way s could be in $sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h])$ but not satisfy $\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h]$ would be for there to exist some path from s where every state on the path was in the set $sat(\mathbf{AX}\ \mathbf{A}[g\ \mathbf{U}\ h]) \cap (S_T - sat(h))$.

To keep such paths from being fair, we require that infinitely often we reach a state in the complement of this set, which is $(S_T - sat(\mathbf{AX}\,\mathbf{A}[g\ \mathbf{U}\ h])) \cup sat(h)$. Thus, we obtain the following acceptance condition:

$$F_T = \{\ ((S_T - sat(\mathbf{AX}\,\mathbf{A}[g\ \mathbf{U}\ h])) \cup sat(h)) \mid \mathbf{AX}\,\mathbf{A}[g\ \mathbf{U}\ h] \in el(f)\ \}.$$

The correctness of the tableau construction is guaranteed by the following lemmas. We state the lemmas (without proof) because of the insight they provide into the tableau construction. We refer the redear to [129] for the proof of a similar theorem. The proof in [129] uses a different notion of fairness from the one used in this chapter, however.

LEMMA 36 For all subformulas g of f, if $s \in sat(g)$, then $s \models_F g$.

The main result of this lemma is that the tableau for f satisfies f. To see this, note that any initial state of $\mathcal{T}_f$ is in $sat(f)$ and therefore every initial state of $\mathcal{T}_f$ satisfies f.

An important property of this tableau construction is that any structure that satisfies f precedes $\mathcal{T}_f$ in the preorder $\preceq_F$. To show this we must define a simulation relation between the tableau and any structure M' that satisfies f. This is achieved by defining two states to be related if and only if they satisfy exactly the same set of subformulas of f. If we define H by

$$H = \{\ (s', s) \mid s = \{\ g \mid g \in el(f), s' \models g\ \}\ \}$$

then this property is guaranteed for the set of elementary formulas $el(f)$. The next lemma proves the property for any subformula of f.

LEMMA 37 If $H(s', s)$, then for every subformula or elementary formula g of f, $s' \models g$ implies $s \in sat(g)$.

The next lemma shows that H is indeed a fair simulation relation between M' and $\mathcal{T}_f$.

LEMMA 38 H is a fair simulation relation between M' and the structure $\mathcal{T}_f$.

THEOREM 18 For any structure M', $M' \models_F f$ if and only if $M' \preceq_F \mathcal{T}_f$.

Proof Suppose $M' \preceq_F \mathcal{T}_f$. By Lemma 36 and the definition of the tableau, every initial state of $\mathcal{T}_f$ satisfies f, i.e., $\mathcal{T}_f \models_F f$. Thus, since $M' \preceq_F \mathcal{T}_f$, $M' \models_F f$.

If $M' \models_F f$, then by definition, every $s'_0 \in S'_0$ satisfies f. Let H be the relation defined above. By the definition of H, every such s'_0 is paired with a (unique) s_0. Lemma 37 implies that $s_0 \in sat(f)$, and by the definition of the tableau, $s_0 \in S_0$. By Lemma 38, H is a simulation relation, so $M' \preceq_F \mathcal{T}_f$. □

The tableau construction can also be used to reason about formulas. We are typically interested in whether every model of a formula g is also a model of some other formula f. Let $g \models f$ denote this semantic relation.

COROLLARY 3 $g \models f$ if and only if $\mathcal{T}_g \models_F f$.

Proof If $g \models f$, then every model of g, in particular $\mathcal{T}_g$, is also a model of f. Assume $\mathcal{T}_g \models_F f$, and let $M \models_F g$. By the previous theorem, $M \preceq_F \mathcal{T}_g$. By Theorem 17, $M \models_F f$. □

12 Compositional Reasoning

Efficient algorithms for compositional verification can extend the applicability of formal verification methods to much larger and more interesting systems. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. If we can deduce that the system satisfies each local property, and if we know that the conjunction of the local properties implies the overall specification, then we can conclude that the complete system satisfies this specification as well.

Consider for instance the problem of verifying a communication protocol that is modeled by three finite state processes: a transmitter, some type of network, and a receiver. Suppose that the specification for the system is that data is eventually transmitted correctly from the sender to the receiver. Such a specification might be decomposed into three local properties. First, the data should eventually be transferred correctly from the transmitter to the network. Second, the data should eventually be transferred correctly from one end of the network to the other. Third, the data should eventually be transferred correctly from the network to the receiver. We might be able to verify the first of these local properties using only the transmitter and the network, the second using only the network, and the third using only the network and the receiver. By decomposing the verification in this way, we never have to compose all of the processes and therefore avoid the state explosion phenomenon.

In this chapter we will focus on the *assume-guarantee paradigm* [115, 150, 198, 218] for compositional reasoning. This technique verifies each component process separately. Suppose that there are two processes M and M'. Since the behavior of process M depends on the behavior of process M', the user specifies a set of assumptions that must be satisfied by M' in order to guarantee the correctness of process M. Since the behavior of process M' also depends on the behavior of process M, the user specifies a set of assumptions that must be satisfied by M in order to guarantee the correctness of process M'. By combining the set of assumed and guaranteed properties of M and M' in an appropriate manner, it is possible to establish correctness of the entire system $M \parallel M'$ without constructing the global state-transition graph.

Typically, a formula is a triple $\langle g \rangle M \langle f \rangle$ where g and f are temporal formulas and M is a program. Although the formula looks like a Hoare triple, it is actually quite different. The formula is true if whenever M is part of a system satisfying the assumption g, the system must also guarantee the property f. A typical proof shows that $\langle g \rangle M' \langle f \rangle$ and $\langle True \rangle M \langle g \rangle$ hold and concludes that $\langle True \rangle M \parallel M' \langle f \rangle$ is true. This proof strategy can also be expressed as an inference rule:

$$\frac{\langle True\rangle M\langle g\rangle \quad \langle g\rangle M'\langle f\rangle}{\langle True\rangle M \parallel M'\langle f\rangle}$$

It is important to avoid circularity in assume-guarantee arguments. Consider the following inference rule:

$$\frac{\langle g\rangle M\langle f\rangle \quad \langle f\rangle M'\langle g\rangle}{M \parallel M' \models f \wedge g}$$

This rule is easily seen to be unsound. Let M be **wait**$(y = 1); x := 1$ and M' be **wait**$(x = 1); y := 1$. Let $g = \mathbf{AF}(y = 1)$ and $f = \mathbf{AF}(x = 1)$. Then, the hypotheses for the proof rule hold while the conclusion does not. Later in this chapter, we will show how to prove the soundness of assume-guarantee inference rules.

There are a number of difficulties involved in developing a verifier that can support this style of reasoning. First, we must be able to check whether *every* system containing a given component satisfies a given local property. Because it is often the case that the local property is only true under certain conditions, we also need to be able to make *assumptions* about the environment of the component when doing the verification. These assumptions, which represent requirements on other components, must be checked in order to complete the verification. In addition, we must provide a method for checking that the conjunction of certain local properties implies a given specification. Tools have been developed by Grumberg and Long and by Josko that permit this type of reasoning to be automated [129, 151, 179].

In the remainder of this section we will describe one possible way to automate this type of reasoning. We use the *fair simulation* preorder $\preceq_F$ and the logic ACTL interpreted over fair Kripke structures. As shown in Theorem 17 in Chapter 11, the preorder $\preceq_F$ has the property that if an ACTL formula is true for a fair structure M', then it is true for any structure M such that $M \preceq_F M'$. We define parallel composition so that a system precedes any component that it contains. That is, $M \parallel M' \preceq_F M$. Moreover, composition with a fixed component preserves the preorder relation between two components. That is, if $M \preceq_F M'$ then $M \parallel M'' \preceq_F M' \parallel M''$. Finally, recall from Section 11.2 that for any ACTL formula f it is possible to construct a special model $\mathcal{T}_f$, called a *tableau*. The tableau $\mathcal{T}_f$ has the property that a structure M satisfies f if and only if M precedes $\mathcal{T}_f$ in the preorder $\preceq_F$. In order to define a tableau with this property, we need to use fairness constraints. This explains why we choose the fair preorder $\preceq_F$ and the

logic *fair* ACTL. In such a framework, the first inference rule in this chapter might be expressed as:

$$\frac{\begin{array}{l} M \preceq_F \mathcal{T}_g, \\ M' \parallel \mathcal{T}_g \models f, \end{array}}{M \parallel M' \models f.}$$

12.1 Composition of Structures

Let $M = (S, S_0, AP, L, R, F)$ and $M' = (S', S'_0, AP', L', R', F')$ be two structures defined as in Chapter 11. The *parallel composition* of M and M', denoted $M \parallel M'$, is the structure M'' defined as follows.

1. $S'' = \{ (s, s') \mid L(s) \cap AP' = L'(s') \cap AP \}$.
2. $S''_0 = (S_0 \times S'_0) \cap S''$.
3. $AP'' = AP \cup AP'$.
4. $L''\big((s, s')\big) = L(s) \cup L'(s')$.
5. $R''\big((s, s'), (t, t')\big)$ if and only if $R(s, t)$ and $R'(s', t')$.
6. $F'' = \{ (P \times S') \cap S'' \mid P \in F \} \cup \{ (S \times P') \cap S'' \mid P' \in F' \}$.

This definition of composition models *synchronous* behavior. States of the composition are pairs of component states that agree on the common atomic propositions. Each transition of the composition involves a joint transition of the two components. The definition is relatively straightforward with the exception of the fairness constraint. The constraint is designed to insure the *fair path property*, which states that a path in $M \parallel M'$ is fair if and only if its restriction to each component results in a fair path. Intuitively, the first set of pairs in the constraint

$$\{ (P \times S') \cap S'' \mid P \in F \}$$

insures that the restriction of a path in M'' to its component in S is a fair path in M. The second set of pairs

$$\{ (S \times P') \cap S'' \mid P' \in F' \}$$

insures that the restriction of the path to its component in S' is a fair path in M'. Because $P \times S'$ and $S \times P'$ may contain pairs that are not states in M'', it is necessary to intersect each with S''.

It is straightforward but tedious to prove that parallel composition is commutative and associative (up to isomorphism). The next three theorems deal with the connection between parallel composition and the simulation preorder $\preceq_F$. The first theorem states that composing M with M' can only restrict the possible behaviors of M. As a consequence of this theorem, it is sufficient to reason about the structure M rather than arbitrary systems containing M. Moreover, this theorem and Theorem 16 imply that a standard CTL model checker [63] can be used to determine if a formula of ACTL is true in all systems containing a given component. This is the key to compositional verification.

THEOREM 19 For all M and M', $M \parallel M' \preceq_F M$.

Proof Let S'' be the set of states of $M \parallel M'$. Define H by

$$H = \{ \big((s, s'), s\big) \mid (s, s') \in S'' \}.$$

If (s_0, s_0') is an initial state of $M \parallel M'$, then $s_0 \in S_0$. The label of (s, s') is $L(s) \cup L'(s')$, and $\big(L(s) \cup L'(s')\big) \cap AP = L(s)$. If $(s_0, s_0'), (s_1, s_1'), \ldots$ is a fair path in $M \parallel M'$, then by the fair path property, $s_0, s_1, \ldots$ is a fair path in M. By the definition of H, we have $H\big((s_i, s_i'), s_i\big)$ for every i. Hence, H is a simulation relation and $M \parallel M' \preceq_F M$. □

The second theorem permits a component of a system to be replaced by an abstraction of that component. Thus, in order to show that some property is true in the system $M \parallel M''$, we can replace M by an abstraction M' and then verify that the property holds in $M' \parallel M''$. Checking $M \preceq_F M'$ insures that M' is indeed an abstraction of M.

THEOREM 20 For all M, M' and M'', if $M \preceq_F M'$ then $M \parallel M'' \preceq_F M' \parallel M''$.

Proof Let H_0 be a simulation relation between M and M'. Define H_1 by

$$H_1 = \{ \big((s, s''), (s', s'')\big) \mid H_0(s, s') \}.$$

It can be shown that H_1 is a simulation relation. □

The last theorem is a technical result that is needed in order to use multiple levels of assume-guarantee reasoning. We will see how it is used in Section 12.2.

THEOREM 21 For all M, $M \preceq_F M \parallel M$.

Proof First note that for every state s of M, (s, s) is a state of $M \parallel M$. Define $H = \{ \big(s, (s, s)\big) \mid s \in S \}$. If $s_0 \in S_0$, then by the definition of composition, (s_0, s_0) is an initial state of $M \parallel M$. (s, s) trivially has the same label as s. Using the fair path property and the definition of composition, we find that if $s_0, s_1, \ldots$ is a fair path in M, then

$(s_0, s_0), (s_1, s_1), \ldots$ is a fair path in $M \parallel M$. By the definition of H, we have $H(s_i, (s_i, s_i))$ for all i. Hence H is a fair simulation relation and $M \preceq_F M \parallel M$. □

12.2 Justifying Assume-Guarantee Proofs

The assume-guarantee paradigm can be applied in many different ways. It is important to know that a particular application of this paradigm is sound. The theory developed earlier in this chapter can be used to justify assume-guarantee proof rules. We illustrate how a rule can be justified by considering an example. We first describe the rule intuitively using an extension of Pnueli's notation [218]. The extension allows assumptions and specifications to be given either as formulas or directly as finite state models, whichever is more concise or convenient.

$$\frac{\begin{array}{l}\langle True\rangle M \langle A\rangle \\ \langle A\rangle M' \langle g\rangle \\ \langle g\rangle M \langle f\rangle\end{array}}{\langle True\rangle M \parallel M' \langle f\rangle}$$

Here, A, M, and M' represent finite state models and g and f represent fair ACTL formulas. In our framework, this corresponds to the proof rule

$$\frac{\begin{array}{l}M \preceq_F A \\ A \parallel M' \models_F g \\ \mathcal{T}_g \parallel M \models_F f\end{array}}{M \parallel M' \models_F f}$$

The soundness of this rule is established by showing that the conclusion must be true if each of the three hypotheses is true.

1.	$M \preceq_F A$	Hypothesis
2.	$M \parallel M' \preceq_F A \parallel M'$	Line (1) and Theorem 20
3.	$A \parallel M' \models_F g$	Hypothesis
4.	$A \parallel M' \preceq_F \mathcal{T}_g$	Line (3) and Theorem 18
5.	$M \parallel M' \preceq_F \mathcal{T}_g$	Lines (2), (4) and Transitivity of $\preceq_F$
6.	$M \parallel M \parallel M' \preceq_F \mathcal{T}_g \parallel M$	Line (5) and Theorem 20
7.	$\mathcal{T}_g \parallel M \models_F f$	Hypothesis
8.	$M \parallel M \parallel M' \models_F f$	Lines (6), (7) and Theorem 17
9.	$M \preceq_F M \parallel M$	Theorem 21

10.	$M \parallel M' \preceq_F M \parallel M \parallel M'$	Line (9) and Theorem 20
11.	$M \parallel M' \models_F f$	Lines (8), (10) and Theorem 17

12.3 Verifying a CPU Controller

A symbolic model checker based on the theory developed earlier in this section is described in Grumberg and Long [129]. It includes facilities for model checking, temporal reasoning (via the tableau construction), and checking if one structure simulates another. The model checker is used to verify a simple CPU controller. We give only a brief description of the CPU here; see [72] for details. The CPU is a simple stack-based machine, that is, part of the CPU's memory contains a stack from which instruction operands are popped and onto which results are pushed. There are two parts to the CPU controller. The first part is called the access unit and is responsible for all the CPU's memory references. The second part, called the execution unit, interprets the instructions and controls the arithmetic unit, shifter, and so on. These two parts operate in parallel. The access unit and execution unit communicate via a small number of signals. Three of the signals, *push*, *pop*, and *fetch*, are inputs of the access unit and indicate that the execution unit wants to push or pop something from the stack or to get the next instruction. For each of these signals there is a corresponding ready output from the access unit. The execution unit must wait for the appropriate ready signal before proceeding. One additional signal, *branch*, is asserted by the execution unit when it wants to jump to a new program location.

In order to increase performance, the access unit attempts to keep the value on the top of the stack in a special register called the TS register. The goal is to keep the execution unit from having to wait for the memory. For example, when the TS register contains valid data, a pop operation can proceed immediately. In addition, when a value is pushed on the stack, it is moved into this register and copied to memory at some later point. The access unit also loads instructions into a queue when possible so that fetches do not require waiting for the memory. This queue is flushed whenever the CPU branches.

The specifications are divided into two classes. The conditions in the first class are safety properties that specify what sequences of operations are allowed. They depend on the access unit asserting the various ready signals at appropriate times and on the memory acknowledge signal being well-behaved. In order to verify the properties, a simple model of the memory was used. By composing this model with the access unit, it was possible to verify all of the properties except one. To verify the remaining property, an additional assumption $\mathbf{AG}(\neg push \vee \neg pop)$ was required. The model checker verified that the property was true under this assumption by building the tableau for the assumption, composing it with the access unit and memory model, and checking the property.

The second class consists of a single liveness property: $\mathbf{AG}\,\mathbf{AF}(\mathit{fetch} \wedge \mathit{fetchrdy})$. This formula states that the CPU infinitely often fetches another instruction. One way to verify this property involves making a model of the execution unit. We describe an alternative way of doing the verification that uses a series of ACTL assumptions.

The idea will be to check the property for the execution unit. In order for the formula to be true, the access unit must eventually respond to push and pop requests and must fill the instruction queue when appropriate. The access unit is guaranteed to meet these conditions provided that the execution unit never tries to do two operations at once and does not remove a request before the corresponding operation is completed. We begin with these properties.

$$\mathbf{AG}\big(\neg(\mathit{fetch} \wedge \mathit{push}) \wedge \neg(\mathit{fetch} \wedge \mathit{pop}) \wedge \cdots \wedge \neg(\mathit{pop} \wedge \mathit{branch})\big) \tag{12.1}$$

$$\mathbf{AG}\big(\mathit{push} \rightarrow \mathbf{A}[\mathit{pushed}\ \mathbf{R}\ \mathit{push}]\big) \tag{12.2}$$

$$\mathbf{AG}\big(\mathit{pop} \rightarrow \mathbf{A}[\mathit{popped}\ \mathbf{R}\ \mathit{pop}]\big) \tag{12.3}$$

The first of these specifies that every pair of operations the execution unit can perform are mutually exclusive. The other two formulas state that if the execution unit makes a push or pop request, then it does not deassert the request until the operation completes. The model checker verified that these properties hold in the execution unit alone, and (using the tableau construction) that the first property implies the assumption $\mathbf{AG}(\neg\mathit{push} \vee \neg\mathit{pop})$ used above. Now using Formulas 12.1 and 12.2 as assumptions, it was possible to check that the system composed of the access unit and the memory model satisfied the formula

$$\mathbf{AG}\big(\mathit{push} \rightarrow \mathbf{A}[\mathit{push}\ \mathbf{U}\ \mathit{pushed}]\big). \tag{12.4}$$

This specification states that every push operation will be completed. Similarly, using Formulas 12.1 and 12.3 as assumptions, it was possible to verify that

$$\mathbf{AG}\big(\mathit{pop} \rightarrow \mathbf{A}[\mathit{pop}\ \mathbf{U}\ \mathit{popped}]\big). \tag{12.5}$$

The system composed of the access unit and the memory model also satisfies the formula $\mathbf{AG}\,\mathbf{AF}(\mathit{fetchrdy} \vee \mathit{branch})$ (at any point, either the access unit will eventually fill the instruction queue or a branch will occur). Finally, using this formula and Formulas 12.4 and 12.5 as assumptions, the model checker verified that the execution unit satisfies $\mathbf{AG}\,\mathbf{AF}(\mathit{fetch} \wedge \mathit{fetchrdy})$. To complete the verification and conclude that the entire specification is true of a system, it is necessary to check that the actual memory is simulated by the model.

13 Abstraction

Abstraction is probably the most important technique for reducing the state explosion problem. In this chapter we describe two different abstraction techniques: The *cone of influence reduction* and *data abstraction*. Both of these techniques are performed on a high level description of the system, before the model for the system is constructed. Thus, we avoid the construction of the unreduced model that might be too big to fit into memory.

The cone of influence reduction attempts to decrease the size of the state transition graph by focusing on the variables of the system that are referred to in the specification. The reduction is obtained by eliminating variables that do not influence the variables in the specification. In this way, the checked properties are preserved, but the size of the model that needs to be verified is smaller.

Data abstraction, on the other hand, involves finding a mapping between the actual data values in the system and a small set of abstract data values. By extending this mapping to states and transitions, it is possible to obtain an abstract system that simulates the original system and is usually much smaller. Because of the reduction in size, it is frequently easier to verify the abstract system than the original system.

13.1 Cone of Influence Reduction

We will show how the cone of influence reduction can be applied to synchronous circuits. Let V be the set of variables of a given circuit. This circuit can be described by a set of equations

$$v_i' = f_i(V),$$

for each $v_i \in V$, where f_i is a boolean function (see also Chapter 2 and Section 6.6.1).

Suppose we are given a set of variables $V' \subseteq V$ that are of interest with respect to the required specification. We would like to simplify the description of the system by referring only to these variables. However, the values of variables in V' might depend on values of variables not in V'. We therefore define the cone of influence C for V' and use C in order to reduce the description of the system.

The cone of influence C of V' is the minimal set of variables such that

- $V' \subseteq C$.
- if for some $v_l \in C$ its f_l depends on v_j, then $v_j \in C$.

We construct a new (reduced) system by removing all the equations whose left hand side variables do not appear in C.

Consider again our example of the modulo 8 counter (Figure 2.1). Its set of equations is

$v_0' = \neg v_0$

$v_1' = v_0 \oplus v_1$

$v_2' = (v_0 \wedge v_1) \oplus v_2$

Clearly, if $V' = \{v_0\}$ then $C = \{v_0\}$, since f_0 does not depend on any variable other than v_0. If $V' = \{v_1\}$ then $C = \{v_0, v_1\}$, since f_1 depends on both of the variables, but $v_2 \notin C$ because no variable in C depends on v_2. Finally, if $V' = \{v_2\}$ then C is the set of all the variables.

More powerful state space-reduction techniques based on the same type of dependency analysis are described in [17] and [162].

We will next show that the cone of influence reduction preserves the correctness of specifications in CTL if they are defined over variables (atomic propositions) in C.

Let $V = \{v_1, \ldots, v_n\}$ be a set of boolean variables and let $M = (S, R, S_0, L)$ be the model of a synchronous circuit defined over V where,

- $S = \{0, 1\}^n$ is the set of all valuations of V.
- $R = \bigwedge_{i=1}^{n} [v_i' = f_i(V)]$.
- $L(s) = \{v_i \mid s(v_i) = 1 \text{ for } 1 \leq i \leq n\}$.
- $S_0 \subseteq S$.

Suppose we reduce the circuit with respect to the cone of influence $C = \{v_1, \ldots, v_k\}$ for some $k \leq n$. The reduced model $\widehat{M} = (\widehat{S}, \widehat{R}, \widehat{S_0}, \widehat{L})$ is defined by

- $\widehat{S} = \{0, 1\}^k$ is the set of all valuations of $\{v_1, \ldots, v_k\}$.
- $\widehat{R} = \bigwedge_{i=1}^{k} [v_i' = f_i(V)]$.
- $\widehat{L}(\widehat{s}) = \{v_i \mid \widehat{s}(v_i) = 1 \text{ for } 1 \leq i \leq k\}$.
- $\widehat{S_0} = \{(\widehat{d_1}, \ldots, \widehat{d_k}) \mid \text{there is a state } (d_1, \ldots, d_n) \in S_0 \text{ such that } \widehat{d_1} = d_1 \wedge \ldots \wedge \widehat{d_k} = d_k\}$.

Let $B \subseteq S \times \widehat{S}$ be the relation defined as follow:

$$((d_1, \ldots, d_n), (\widehat{d_1}, \ldots, \widehat{d_k})) \in B \iff d_i = \widehat{d_i} \text{ for all } 1 \leq i \leq k$$

We show that B is a bisimulation relation between M and $\widehat{M}$. First note that for every initial state in S there is a corresponding initial state in $\widehat{S}$ and vice versa. Let $s = (d_1, \ldots, d_n)$ and $\widehat{s} = (\widehat{d_1}, \ldots, \widehat{d_k})$ such that $(s, \widehat{s}) \in B$. Then $d_i = \widehat{d_i}$ for every $1 \leq i \leq k$. Thus, their labeling restricted to $C = \{v_1, \ldots, v_k\}$ agree, that is,

$$L(s) \cap C = \widehat{L}(\widehat{s}).$$

Let $s \to t$ be a transition in M. We show that there is a transition $\widehat{s} \to \widehat{t}$ in $\widehat{M}$ such that $(t, \widehat{t}) \in B$. Denote $t = (e_1, \ldots, e_n)$.

The definition of R implies that for every $1 \leq i \leq n$, $v_i' = f_i(V)$. However, for $1 \leq i \leq k$, v_i depends only on variables in C, hence $v_i' = f_i(C)$. Furthermore, $(s, \widehat{s}) \in B$ implies $\bigwedge_{i=1}^{k} (d_i = \widehat{d_i})$. Thus, for every $1 \leq i \leq k$,

$$e_i = f_i(d_1, \ldots, d_k) = f_i(\widehat{d_1}, \ldots, \widehat{d_k}).$$

If we choose $\widehat{t} = (e_1, \ldots, e_k)$ then $\widehat{s} \to \widehat{t}$ and $(t, \widehat{t}) \in B$ as required.

Now let $\widehat{s} \to \widehat{t}$ be a transition in $\widehat{R}$ where $\widehat{t} = (\widehat{e_1}, \ldots, \widehat{e_k})$. Then, for every $1 \leq i \leq k$, $\widehat{e_i} = f_i(\widehat{d_1}, \ldots, \widehat{d_k})$. Consider the transition $s \to t$ in R for some $t = (e_1, \ldots, e_n)$. Since $\bigwedge_{i=1}^{k} (d_i = \widehat{d_i})$ and since the value of $v_i \in C$ depends only on values of variables in C, we have:

$$\widehat{e_i} = f_i(\widehat{d_1}, \ldots, \widehat{d_k}) = f_i(d_1, \ldots, d_k) = f_i(d_1, \ldots, d_k, d_{k+1}, \ldots d_n) = e_i.$$

Hence, $(t, \widehat{t}) \in B$. This completes the proof that B is a bisimulation between M and $\widehat{M}$. Thus, $M \equiv \widehat{M}$.

The following theorem is a direct consequence of the above result and of Theorem 14 in Chapter 11:

THEOREM 22 Let f be a CTL* formula with atomic propositions in C. Then $M \models f \Leftrightarrow \widehat{M} \models f$.

13.2 Data Abstraction

Verification techniques based on abstraction appear to be necessary for reasoning about circuits that contain data paths or concurrent programs that contain complex data structures. Traditionally, finite- state verification methods have been used mainly for control-oriented systems. The symbolic methods make it possible to handle some systems that involve nontrivial data manipulation, but the complexity of verification is often high. Data abstraction is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. For example, in verifying the addition operation of a microprocessor, it may be necessary to require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations *abstraction* can be used to reduce the complexity of model checking. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The

abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level. Clarke, Grumberg, and Long prove that under such a mapping any properties expressible in the logic ACTL that are satisfied by the abstract system will also be true of the actual system [69, 179].

In order to use this technique in practice, we must be able to construct the OBDD for an abstract model directly without first building the complete model. This can be achieved by starting with a high-level description of the model and combining the abstraction process with compilation. In Chapter 2 a high-level description has been introduced by means of first order formulas $\mathcal{S}_0$ and $\mathcal{R}$ that represent the set of initial states and the transition relation for the system. There, atomic propositions have the form $x = d$, for a system variable x ranging over some domain D and for a value $d \in D$.

Unfortunately, owing to the large (or perhaps even infinite) state space of interesting programs, it may not be possible to apply model checking techniques directly. Furthermore, if the exact values of all the program variables are visible, state space reduction techniques are ineffective. Thus, it may be necessary to hide some of this information. For this reason, it is convenient to map the possible values for each program variable into a small number of abstract values. For example, suppose x is a variable and the domain D_x is the set of all integers. Assume that we are interested in expressing a property involving the sign of x. We create a domain A_x of abstract values for x, with $A_x = \{a_0, a_+, a_-\}$, and define a mapping h_x from D_x to A_x as follows:

$$h_x(d) = \begin{cases} a_0, & \text{if } d = 0, \\ a_+, & \text{if } d > 0, \text{ and} \\ a_-, & \text{if } d < 0. \end{cases}$$

Now the abstract value of x can be expressed using just three atomic propositions. These propositions will be denoted by '$\widehat{x} = a_0$', '$\widehat{x} = a_+$', and '$\widehat{x} = a_-$', where $\widehat{x}$ denotes a reference to an abstract value rather than an actual value. Note that it may no longer be possible to express properties about the exact actual value of x using these atomic propositions. In many cases though, by judicious choice of the abstraction mapping, knowing just the abstract value is sufficient. If this abstraction process is applied to some of the program variables, the Kripke structure obtained will have a smaller number of atomic propositions. Now state space-reduction techniques may be applicable to reduce the complexity of verification.

The particular technique that will be used is based on simulation relations, presented in Chapter 11. The idea will be to merge together all states that have the same labeling of (abstract-level) atomic propositions. Thus, in the reduced structure, every state will have a unique labeling. Consequently, the labeling can be used to identify the state. In the previous example, all states labeled with '$\widehat{x} = a_+$' are collapsed into one state, that is, all states

where $x > 0$ are merged into one. The collapsed state is denoted by the label '$\widehat{x} = a_+$' as well. When the collapsing is performed, it is necessary to ensure that the reduced structure simulates the original one. So, if there is a transition in M between states corresponding to $x = 0$ and $x = 5$, in the reduced system it is necessary to add a transition between the states labeled with '$\widehat{x} = a_0$' and '$\widehat{x} = a_+$'. Similarly, if the state where $x = -7$ is an initial state, the state labeled with '$\widehat{x} = a_-$' will be an initial state.

Later, we will describe more formally how a reduced structure can be obtained from a given structure. The procedure that we describe produces an "ideal" abstraction. It is too inefficient to be used in practice, but explains the concepts that form the basis for our abstraction technique. Later, we will explain how this technique can be implemented efficiently.

Suppose that we have a structure with variables that range over a set of values D. In order to construct the reduced structure, we first change the labeling of the original structure. This is done by choosing an abstract domain A and a mapping h from D to A. This determines a set of abstract atomic propositions AP as described informally above. We now obtain a new structure $M = (S, R, S_0, L)$ that is identical to the original one except that L labels each state with a set of abstract atomic propositions from AP. The structure M can be collapsed into a reduced structure M_r defined as follows:

1. $S_r = \{ L(s) \mid s \in S \}$. Thus, the set of states in the reduced structure is the set of all labelings of states of M.
2. $s_r \in S_0^r$ if and only if there exists s such that $s_r = L(s)$ and $s \in S_0$.
3. $AP_r = AP$.
4. Each s_r is just a set of atomic propositions, so $L_r(s_r) = s_r$.
5. $R_r(s_r, t_r)$ if and only if there exist s and t such that $s_r = L(s)$, $t_r = L(t)$, and $R(s, t)$.

We think of M_r as an abstract version of the Kripke structure of the program. Each abstract state represents a set of concrete states that are merged together during the collapsing process. Note that M_r is completely determined by the choice of the mapping h and the set of abstract values A. If a different mapping or a different set of abstract values is chosen, then a different reduced structure is obtained. It is now easy to see that the reduced structure M_r simulates the original structure M, because $H = \{ (s, s_r) \mid s_r = L(s) \}$ can be used as a simulation relation. Thus, whatever ACTL* properties we can prove about M_r will also hold in M. Note that by using this technique it is only possible to determine whether formulas over the abstract atomic propositions AP are true in M. In practice, AP is chosen by the user so that it is possible to express the properties of M that need to be checked.

Figure 13.1 illustrates the abstraction procedure for a simple traffic light controller. The original program has one variable *color* that can take on values from the set $D =$

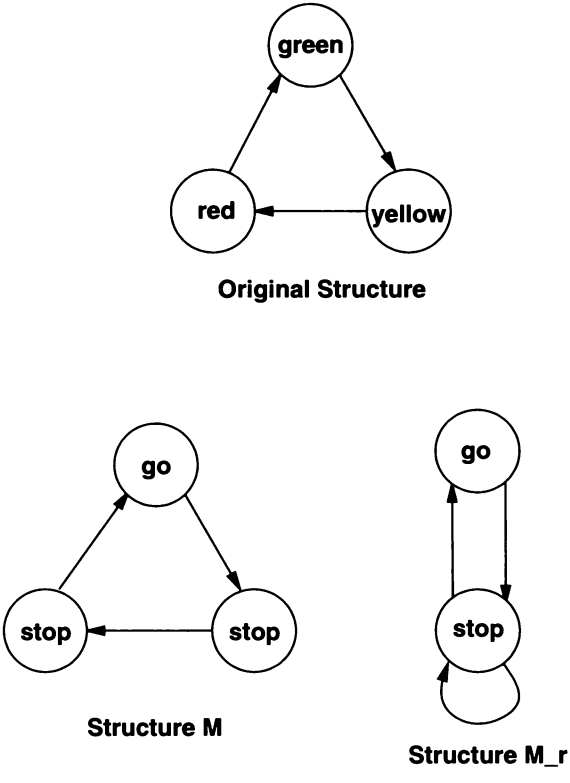


Figure 13.1
Traffic light-controller example.

$\{red, yellow, green\}$. Its states are labeled with atomic propositions '$color = red$', '$color = yellow$', and '$color = green$', which we abbreviate in the figure by *red*, *yellow* and *green*, respectively. The structure M is obtained by choosing an abstract set of values $A = \{stop, go\}$ and a mapping function h defined by:

$$h(red) = stop \qquad h(yellow) = stop \qquad h(green) = go.$$

The set of abstract atomic propositions is given by

$$AP = \{\text{`}\widehat{color} = stop,\text{'} \ \ \text{`}\widehat{color} = go\text{'}\}.$$

In the figure we use *stop* and *go* to abbreviate these atomic propositions. The reduced structure M_r results from merging together those states of M with the same labeling of abstract atomic propositions.

As described above M_r can be used to deduce properties of the program because M_r simulates M. The main difficulty is that building M_r requires constructing M. When it is impossible to build M directly, we use an implicit representation of M in terms of the formulas $\mathcal{S}_0$ and $\mathcal{R}$. Instead of constructing M, the reduced structure M_r is derived from these formulas. In many cases, M_r will still be too large to construct exactly. Thus, an approximation M_a that simulates M_r is constructed. Our goal is to have M_a sufficiently close to M_r so that it is still possible to verify interesting properties of the program.

13.2.1 Computing Approximations

Throughout this section, assume that ϕ, ϕ_1, and ϕ_2 are first order formulas constructed from the primitive relations representing the operations in the program. For simplicity, assume that all of the variables $x_1, x_2, \ldots,$ range over the same domain D. We use a set $\widehat{x_1}, \widehat{x_2}, \ldots,$ of variables ranging over an abstract domain A, with $\widehat{x_i}$ representing the abstract value of x_i. We will also assume that there is only one abstraction function h, which is a surjection, mapping the elements of D *onto* elements of A.

We use the first order formulas $\mathcal{S}_0$ and $\mathcal{R}$ described in Chapter 2 to define the Kripke structure $M = (S, R, S_0, L)$ with state set $S = D \times \cdots \times D$. S_0 is the set of valuations that satisfy the formula $\mathcal{S}_0$. Similarly, the transition relation R is derived from the formula $\mathcal{R}$. The labeling function L is defined over the abstract atomic propositions as follows. Let $s = (d_1, \ldots, d_n)$, that is, in state s, x_i has value d_i. Define $a_i = h(d_i)$. We introduce an atomic proposition '$\widehat{x_i} = a_i$' to denote that x_i has the abstract value a_i. Now $L(s) = \{$'$\widehat{x_1} = a_1$', $\ldots$, '$\widehat{x_n} = a_n$'$\}$.

To produce M_r over the abstract state set $A \times \cdots \times A$ we construct formulas over $\widehat{x_1}, \ldots, \widehat{x_n}$ and $\widehat{x_1}', \ldots, \widehat{x_n}'$ that will represent the initial states and the transition relation of M_r. First, note that it is possible to obtain an abstract Kripke structure M_r by evaluating the formulas

$$\widehat{\mathcal{S}_0} = \exists x_1 \ldots \exists x_n \, (h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_n) = \widehat{x_n} \wedge \mathcal{S}_0(x_1, \ldots, x_n)).$$

and

$$\widehat{\mathcal{R}} = \exists x_1 \ldots \exists x_n \exists x_1' \ldots \exists x_n' \, (h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_n) = \widehat{x_n}$$
$$\wedge\, h(x_1') = \widehat{x_1}' \wedge \cdots \wedge h(x_n') = \widehat{x_n}' \wedge \mathcal{R}(x_1, \ldots, x_n, x_1', \ldots, x_n')).$$

For conciseness, this existential abstraction operation is denoted by $[\cdot]$. If ϕ depends on the free variables $x_1, \ldots, x_m$, then define

$$[\phi](\widehat{x_1}, \ldots, \widehat{x_m}) = \exists x_1 \ldots \exists x_m \, (h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_m) = \widehat{x_m} \wedge \phi(x_1, \ldots, x_m)).$$

Note that the free variables of $[\phi]$ are the abstract versions of $x_1, \ldots, x_m$. So the abstract structure corresponding to M_r is given by the formulas $\widehat{\mathcal{S}}_0 = [\mathcal{S}_0]$ and $\widehat{\mathcal{R}} = [\mathcal{R}]$.

Ideally, we would like to extract S_0^r and R_r from $[\mathcal{S}_0]$ and $[\mathcal{R}]$. However, this is often computationally expensive. To circumvent this difficulty we will define a transformation $\mathcal{A}$ on formula ϕ. The idea of $\mathcal{A}$ is to simplify the formulas to which $[\cdot]$ is applied. This will make it easier to extract the Kripke structure from the formulas. Assume that ϕ is given in negation normal form, that is, negations are applied only to primitive relations.

1. $\mathcal{A}(P(x_1, \ldots, x_m)) = [P](\widehat{x_1}, \ldots, \widehat{x_m})$ if P is a primitive relation.
Similarly, $\mathcal{A}(\neg P(x_1, \ldots, x_m)) = [\neg P](\widehat{x_1}, \ldots, \widehat{x_m})$.
2. $\mathcal{A}(\phi_1 \wedge \phi_2) = \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$.
3. $\mathcal{A}(\phi_1 \vee \phi_2) = \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$.
4. $\mathcal{A}(\exists x\ \phi) = \exists \widehat{x}\ \mathcal{A}(\phi)$.
5. $\mathcal{A}(\forall x\ \phi) = \forall \widehat{x}\ \mathcal{A}(\phi)$.

In other words, $\mathcal{A}$ pushes the existential quantifications inward so that the abstraction operation $[\cdot]$ is only applied at the innermost level. Because these inner abstractions are relatively simple, they can be evaluated easily. Thus, although we may not be able to evaluate $[\mathcal{S}_0]$ and $[\mathcal{R}]$, we generally can evaluate $\mathcal{A}(\mathcal{S}_0)$ and $\mathcal{A}(\mathcal{R})$. This will yield the Kripke structure $M_a = (S_a, S_0^a, R_a, L_a)$. L_a is defined as follows. Let $s_a = (a_1, \ldots, a_n) \in S_a$. Then $L_a(s_a) = \{\text{`}\widehat{x_1} = a_1\text{'}, \ldots, \text{`}\widehat{x_n} = a_n\text{'}\}$. Note that $s = (d_1, \ldots, d_n) \in S$ and s_a will be identically labeled if for all i, $h(d_i) = a_i$.

The price paid for simplifying the evaluation is that it may be necessary to add extra initial states and transitions to the corresponding structure. This is because $[\phi]$ implies $\mathcal{A}(\phi)$ but is not necessarily equivalent to it. As a result, M_a will only be an approximation to M_r. On the other hand, in order to know that M_a simulates M_r, we must show that applying $\mathcal{A}$ cannot cause us to lose any initial states or transitions. This is a consequence of the following theorem.

THEOREM 23 $[\phi]$ implies $\mathcal{A}(\phi)$. In particular, $[\mathcal{S}_0] \rightarrow \mathcal{A}(\mathcal{S}_0)$ and $[\mathcal{R}] \rightarrow \mathcal{A}(\mathcal{R})$.

Proof We apply induction on the structure of the formula ϕ.

1. If $\phi = P(x_1, \ldots, x_m)$ or $\phi = \neg P(x_1, \ldots, x_m)$ where P is a primitive relation then $[\phi] = \mathcal{A}(\phi)$ and the theorem holds.
2. Let $\phi(x_1, \ldots, x_m) = \phi_1 \wedge \phi_2$. Then, $[\phi_1 \wedge \phi_2]$ is identical to the formula

$$\exists x_1 \ldots \exists x_m \left(\bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_1 \wedge \phi_2\right).$$

This formula implies (but is not equivalent to)

$$\exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_1) \wedge \exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_2),$$

which is exactly $[\phi_1] \wedge [\phi_2]$. By the definition of $\mathcal{A}$, $\mathcal{A}(\phi_1 \wedge \phi_2) = \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$, and by the induction hypothesis we have $[\phi_1]$ implies $\mathcal{A}(\phi_1)$ and $[\phi_2]$ implies $\mathcal{A}(\phi_2)$. Hence $[\phi_1] \wedge [\phi_2]$ implies $\mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$, and so $[\phi_1 \wedge \phi_2]$ implies $\mathcal{A}(\phi_1 \wedge \phi_2)$.

3. The case where $\phi = \phi_1 \vee \phi_2$ is similar to the previous case.
4. Let $\phi(x_1, \ldots, x_m) = \forall x \; \phi_1$. Then $[\forall x \; \phi_1]$ is

$$\exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x_i} \wedge \forall x \; \phi_1(x, x_1, \ldots, x_m)).$$

Assume without loss of generality that the bound variable x is different from the x_i and $\widehat{x_i}$. Then, the above formula is equivalent to

$$\exists x_1 \ldots \exists x_m \forall x \, (\bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_1(x, x_1, \ldots, x_m)).$$

This implies (but is not equivalent to)

$$\forall x \exists x_1 \ldots \exists x_m \, (\bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_1(x, x_1, \ldots, x_m)).$$

Because h is a surjection, for every abstract element in A, there is some element of D that maps onto it. Hence the above formula implies

$$\forall \widehat{x} \exists x [\exists x_1 \ldots \exists x_m \, (h(x) = \widehat{x} \wedge \bigwedge_i h(x_i) = \widehat{x_i} \wedge \phi_1(x, x_1, \ldots, x_m))].$$

This is exactly $\forall \widehat{x} \, [\phi_1]$. Now by the induction hypothesis, $[\phi_1]$ implies $\mathcal{A}(\phi_1)$, and so $\forall \widehat{x} \, [\phi_1]$ implies $\forall \widehat{x} \; \mathcal{A}(\phi_1)$. This latter formula is equal to $\mathcal{A}(\forall x \; \phi_1)$.

5. The case where $\phi = \exists x \; \phi_1$ is similar to the previous case. □

The above idea of "pushing the abstractions inward" is essentially the same one that is used in abstract interpretation [20, 86, 87, 89, 91, 201, 203]. By defining a suitable abstract domain of computation and then interpreting a program relative to this domain, it is possible to extract information about the program. The goal is usually to obtain data that will be used at compile time to optimize the program. Abstract interpretations have been defined for applications such as: strictness and reference count analysis for functional programs; finding linear relationships between variables; and computing live ranges for

variables. The interpretation is done using abstract versions of the language operators. These abstract operators correspond to the abstract versions of the primitive relations above. The relationships between abstract interpretation and the use of abstraction in model checking are discussed in more detail elsewhere [20, 89, 91].

Finally, we show that M_a simulates M. This is the basis for using abstraction to verify properties of the program.

THEOREM 24 $M \preceq M_a$.

Proof We show this by giving a simulation relation between M and M_a. Let $s = (d_1, \ldots, d_n)$ and $s_a = (a_1, \ldots, a_n)$. Define $H(s, s_a)$ if and only if for all i, $h(d_i) = a_i$.

Assume $H(s, s_a)$, with $s = (d_1, \ldots, d_n)$. Then $s_a = (h(d_1), \ldots, h(d_n))$. Note first that the two states have the same labeling. Recall that the label of s is the set of propositions '$\widehat{x_i} = a_i$,' where the value of x_i is mapped to a_i by h. Assume then that '$\widehat{x_i} = a_i$' labels s. This is true if and only if $a_i = h(d_i)$. Now, s_a will be labeled by '$\widehat{x_i} = a_i$' if and only if the ith component of s_a is a_i. But the ith component of s_a is $h(d_i)$, which is equal to a_i.

Assume $R(s, t)$, where $t = (e_1, \ldots, e_n)$. Define $t_a = (h(e_1), \ldots, h(e_n))$. We must show that $R_a(s_a, t_a)$. By the definition of R, it is known that s and t correspond to valuations satisfying $\mathcal{R}$. Now we show that $[\mathcal{R}](s_a, t_a)$. By definition of $[\cdot]$, $[\mathcal{R}](s_a, t_a)$ holds if and only if

$$\exists x_1 \ldots \exists x_n \exists x'_1 \ldots \exists x'_n \left(\bigwedge_{i=1}^{n} (h(x_i) = h(d_i) \wedge h(x'_i) = h(e_i)) \right.$$
$$\left. \wedge\ \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n)\right).$$

Because $\mathcal{R}(s, t)$, the above formula holds by taking the d_i as witnesses for the x_i and the e_i as witnesses for the x'_i. Now that we know that $[\mathcal{R}](s_a, t_a)$ is true, the previous theorem implies that $\mathcal{A}(\mathcal{R})(s_a, t_a)$ is true. But $\mathcal{A}(\mathcal{R})$ defines R_a, so $R_a(s_a, t_a)$ holds. Thus, H is a simulation relation between M and M_a.

Using a similar argument, we see that if $s \in S_0$, then $s_a \in S_0^a$. Thus every initial state of M has a corresponding initial state of M_a, and so $M \preceq M_a$. □

13.2.2 Exact Approximations

In the previous section, it was shown that $M \preceq M_a$; thus, every ACTL* formula satisfied by M_a also holds in M. In this section, we consider some additional conditions that allow us to show that M is bisimulation equivalent to M_a. Based on the results of Chapter 11, these conditions permit us to verify any CTL* formula using M_a and conclude that M also satisfies the formula. When these conditions are satisfied, M_a will be called an *exact approximation* of M.

To begin, note that each abstraction mapping h_x for the program variable x induces an equivalence relation $\sim_x$ defined as follows. Let d_1 and d_2 be in D_x. Then $d_1 \sim_x d_2$ if and only if $h_x(d_1) = h_x(d_2)$. The key condition for exact approximations is that these equivalence relations are *congruences* for the primitive relations corresponding to the basic operations used in the program. Recall the definition of congruence: Let $P(x_1, \ldots, x_m)$ be a relation with x_i ranging over D_{x_i}. The equivalence relations $\sim_{x_i}$ are a congruence with respect to P if and only if

$$\forall d_1 \ldots \forall d_m \forall e_1 \ldots \forall e_m \, (\bigwedge_{i=1}^{m} d_i \sim_{x_i} e_i \rightarrow (P(d_1, \ldots, d_m) \Leftrightarrow P(e_1, \ldots, e_m))).$$

The proof of exactness when the $\sim_{x_i}$ are congruences with respect to the primitive relations is based on the following theorem which is the analog of Theorem 23. This theorem allows us to show that M_r and M_a are bisimulation equivalent.

THEOREM 25 If the $\sim_{x_i}$ are congruences with respect to the primitive relations and ϕ is a formula defined over these relations, then $[\phi] \Leftrightarrow \mathcal{A}(\phi)$. In particular $[\mathcal{S}_0]$ and $[\mathcal{R}]$ are equivalent to $\mathcal{A}(\mathcal{S}_0)$ and $\mathcal{A}(\mathcal{R})$, respectively.

Clarke, Grumberg, and Long prove bisimulation equivalence between M and M_a.

THEOREM 26 If $\sim_{x_i}$ are congruences with respect to the primitive relations, then $M \equiv M_a$.

For the proofs of these theorems, see [69].

13.2.3 A Simple Language

We now describe a verification system based on the ideas in the previous sections. The system consists of a compiler for a finite state language, plus an OBDD-based model checker. This section contains a brief discussion of the language, which is designed for specifying reactive programs. The main features of this language are:

1. It is procedural and contains a variety of structured programming constructs, such as **while** loops. Nonrecursive procedures are also available.

2. It is finite state. The user must specify a fixed number of bits for each input and output in a program.

3. The model of computation is a synchronous one. At the start of each time step, inputs to the program are obtained from the environment. All computation in a program is viewed as instantaneous (i.e., occurring in zero time). There is one special statement, **wait**, which is used to indicate the passage of time. When a **wait** statement is encountered, any changes to the program's outputs become visible to the environment, and a new time step is initiated.

Thus, computation proceeds as follows: obtain inputs, compute (in zero time) until a **wait** is encountered, make output changes visible, obtain new inputs, and so on. The **wait** statements indicate the control points in the program. The semantics of **wait** statements used here is common in synchronous programming languages. Note that it is different from the semantics used for shared memory, asynchronous programs in Chapters 2 and 12.

Aside from the **wait** statement, most of the language features used in the examples in this chapter are self-explanatory.

A program in the language may be compiled into a Moore machine [200] for implementation in hardware. A Moore machine is a standard model for synchronous circuits, and for verification there is a standard transformation of Moore machines into Kripke structures. When abstraction is not used, our compiler produces this Kripke structure directly. Because the structure may have a large number of states, it is important not to generate an explicit-state representation. Instead, the compiler directly produces a description of the structure in the form of an OBDD. This is then used as the input to the model checking program.

The user specifies abstractions for some of the variables at the time of compilation. By employing the techniques described in the previous sections, the compiler directly generates an abstract structure. There are a number of abstractions built into the compiler, some of which are described in the following section. In addition, the user may define new abstractions by supplying procedures to build the OBDDs representing them. Abstract versions of the primitive relations are computed automatically by the compiler.

Figure 13.2 is a small example program, a settable countdown timer, written in the language. The timer has two input variables, *set* and *start*, which are one and eight bits wide, respectively. There are also two output variables: *count*, which is eight bits wide and is initially zero; and *alarm*, which is one bit and initially one. At each time step, the operation of the counter is as follows. If *set* is one, then the counter is set to the value of *start*. Otherwise, if the counter is not zero, it is decremented. The *alarm* output is set to one when *count* is zero, and to zero if *count* is nonzero.

13.2.4 Example Abstractions

In this section we discuss some abstractions that have proved useful in practice. Each is illustrated with a small example. The temporal logic formulas in this section are written with some syntactic sugaring of the atomic propositions in order to make them easier to read. For example, if x is a variable that is abstracted by:

$$h(d) = \begin{cases} a_{\text{even}}, & \text{if } d \text{ is even;} \\ a_{\text{odd}}, & \text{if } d \text{ is odd,} \end{cases}$$

then we will generally write $even(x)$ in a formula rather than '$\widehat{x} = a_{\text{even}}$'.

```
input set : 1;
input start : 8;
output count : 8 := 0;
output alarm : 1 := 1;

loop
    if set = 1 then
        count := start;
    else if count > 0 then
        count := count − 1;
    end if;
    if count = 0 then
        alarm := 1;
    else
        alarm := 0;
    end if;
    wait;
end loop;
```

Figure 13.2
An example program.

13.2.5 Congruence Modulo an Integer

For verifying programs involving arithmetic operations, a useful abstraction is congruence modulo a specified integer m:

$h(i) = i \bmod m$.

This abstraction is motivated by the following properties of arithmetic modulo m.

$$\big((i \bmod m) + (j \bmod m)\big) \bmod m \equiv i + j \pmod m$$

$$\big((i \bmod m) - (j \bmod m)\big) \bmod m \equiv i - j \pmod m$$

$$\big((i \bmod m)(j \bmod m)\big) \bmod m \equiv ij \pmod m$$

In other words, we can determine the value modulo m of an expression involving addition, subtraction, and multiplication by working with the values modulo m of the subexpressions.

The abstraction may also be used to verify more complex relationships by applying the following result from elementary number theory.

THEOREM 27 (Chinese remainder theorem) Let $m_1, m_2, \ldots, m_n$ be positive integers that are pairwise relatively prime. Define $m = m_1 m_2 \ldots m_n$, and let $b, i_1, i_2, \ldots, i_n$ be integers. Then there is a unique integer i such that

$$b \leq i < b + m \quad \text{and} \quad i \equiv i_j \pmod{m_j} \quad \text{for } 1 \leq j \leq n.$$

Suppose that we are able to verify that at a certain point in the execution of a program, the value of the nonnegative integer variable x is equal to i_j modulo m_j for each of the relatively prime integers $m_1, m_2, \ldots, m_n$. Further, suppose that the value of x is constrained to be less than $m_1 m_2 \ldots m_n$. Then using the above result, it is possible to conclude that the value of x at that point in the program is uniquely determined.

We illustrate this abstraction using a 16 bit by 16 bit unsigned multiplier (see Figure 13.3). The program has inputs *req*, *in1*, and *in2*. The last two inputs provide the factors to operate on, and the first is a request signal that starts the multiplication. Some number of time units later, the output *ack* will be set to true. At that point, either *output* gives the 16 bit result of the multiplication, or *overflow* is one if the multiplication overflowed. The multiplier then waits for *req* to become zero before starting another cycle. The multiplication itself is done with a series of shift-and-add steps. At each step, the low-order bit (bit 0) of the first factor is examined; if it is one, then the second factor is added to the accumulating result. The first factor is then shifted right and the second factor is shifted left in preparation for the next step.

A feature of the language that the program uses is the ability to extend an operand to a specified number of bits. For example, x: 5 extends x to be 5 bits wide by adding leading 0 bits. This facility is used to extend *output* and *factor2* when adding and shifting so that overflow can be detected. The statement (*overflow*, *output*) := (*output*: 17) + *factor2* sets *output* to the 16-bit sum of *output* and *factor2* and *overflow* to the carry from this sum. Also, $x \ll 1$ is x shifted left by one bit. Right shifts are indicated using $\gg$. The **break** statement is used to exit the innermost loop.

The specification for the multiplier is a series of formulas of the following form.

$$\mathbf{AG}\big(waiting \wedge req \wedge (in1 \bmod m = i) \wedge (in2 \bmod m = j)$$
$$\rightarrow \mathbf{A}[\neg ack\ \mathbf{U}\ ack \wedge (overflow \vee (output \bmod m = ij \bmod m))]\big)$$

Here, i and j range from 0 through $m - 1$, and *waiting* is an atomic proposition that is true when execution is at the program statement labeled 1. Note that this specification admits

```
input in1 : 16;
input in2 : 16;
input req : 1;
output factor1 : 16 := 0;
output factor2 : 16 := 0;
output output : 16 := 0;
output overflow : 1 := 0;
output ack : 1 := 0;

procedure waitfor(e)
    while ¬e
        wait;
    end while;
end procedure;

loop
  1: waitfor(req);
     factor1 := in1;
     factor2 := in2;
     output := 0;
     overflow := 0;
     wait;
     loop
         if (factor1 = 0) ∨ (overflow = 1) then break;
         if lsb(factor1) = 1 then
             (overflow, output) := (output: 17) + factor2;
         factor1 := factor1 ≫ 1;
         wait;
         if (factor1 = 0) ∨ (overflow = 1) then break;
         (overflow, factor2) := (factor2: 17) ≪ 1;
         wait;
     end loop;
     ack := 1;
     wait;
     waitfor(¬req);
     ack := 0;
end loop;
```

Figure 13.3
A 16-bit multiplier.

the possibility that the multiplier always signals an overflow. We will verify that this is not the case using a different abstraction (see Section 13.2.6).

The input *in2* and the outputs *factor2* and *output* were all abstracted modulo m. The output *factor1* was not abstracted because its entire bit pattern is used to control when *factor2* is added to *output*. The verification was performed for $m = 5$, 7, 9, 11, and 32. These numbers are relatively prime, and their product, 110,880, is sufficient to cover all 2^{16} possible values of *output*. The entire verification required slightly less than thirty minutes of CPU time on a Sun 4. Checking the above formulas on the unabstracted multiplier proved to be impractical.

13.2.6 Representation by Logarithm

When only the order of magnitude of a quantity is important, it is sometimes useful to represent the quantity by (a fixed precision approximation of) its logarithm. For example, suppose $i \geq 0$. Define

$$\lg i = \lceil \log_2(i+1) \rceil,$$

that is, $\lg i$ is 0 if i is 0, and for $i > 0$, $\lg i$ is the smallest number of bits needed to write i in binary. We take $h(i) = \lg i$.

As an illustration of this abstraction, consider again the multiplier of Figure 13.3. Recall that a program that always indicated an overflow would satisfy our previous specification. Note that if $\lg i + \lg j \leq 16$, then $\lg ij \leq 16$, and hence the multiplication of i and j should not overflow. Conversely, if $\lg i + \lg j \geq 18$, then $\lg ij \geq 17$, and the multiplication of i and j will overflow. When $\lg i + \lg j = 17$, it is impossible to say whether overflow should occur. These observations lead us to strengthen our specification to include the following two formulas.

$$\mathbf{AG}\big(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \leq 16) \rightarrow \mathbf{A}[\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge \neg \mathit{overflow}]\big)$$

$$\mathbf{AG}\big(\mathit{waiting} \wedge \mathit{req} \wedge (\lg \mathit{in1} + \lg \mathit{in2} \geq 18) \rightarrow \mathbf{A}[\neg \mathit{ack}\ \mathbf{U}\ \mathit{ack} \wedge \mathit{overflow}]\big)$$

All of the 16-bit variables in the program are represented by their logarithms. Compiling the program with this abstraction and checking the above properties required less than a minute of CPU time.

13.2.7 Single Bit and Product Abstractions

For programs involving bitwise logical operations, the following abstraction is often useful:

$$h(i) = \text{the } j\text{th bit of } i,$$

```
input in : 16;
output parity : 1 := 0;
output b : 16 := 0;
output done : 1 := 0;

b := in;
wait;
while b ≠ 0 do
    parity := parity ⊕ lsb(b);
    b := b ≫ 1;
    wait;
end while;
done := 1;
```

Figure 13.4
A parity computation program.

where j is some fixed number.

If h_1 and h_2 are abstraction mappings, then

$$h(i) = \big(h_1(i), h_2(i)\big)$$

also defines an abstraction mapping. Using this abstraction, it may be possible to verify properties that it is not possible to verify with either h_1 or h_2 alone.

As an example of using these types of abstractions, consider the program shown in Figure 13.4. This program reads an initial 16-bit input and computes the parity of it. The output *done* is set to one when the computation is complete; at that point, *parity* has the result. Let $\sharp i$ be true if the parity of i is odd. One desired property of the program is the following.

1. The value assigned to b has the same parity as that of *in*; and
2. $\sharp b \oplus parity$ is invariant from that point onward.

We can express the above with the following formula.

$$\neg\sharp in \wedge \mathbf{AX}\big(\neg\sharp b \wedge \mathbf{AG}\,\neg(\sharp b \oplus parity)\big) \vee \sharp in \wedge \mathbf{AX}\big(\sharp b \wedge \mathbf{AG}(\sharp b \oplus parity)\big)$$

To verify this property, a combined abstraction for *in* and b was used. Namely, the possible values for these variables were grouped both by the value of their low-order bit and by their parity. The verification required only a few seconds.

```
input a : 8;
output b : 8 := 0;

loop
    b := a;
    wait;
end loop;
```

Figure 13.5
A simple program.

13.2.8 Symbolic Abstractions

The use of an OBDD-based compiler together with a model checker makes it possible to use abstractions that depend on symbolic values. This idea can greatly increase the power of a particular type of abstraction. As a simple example, consider the program in Figure 13.5. We wish to show that the next state value of b is always equal to the current state value of a. This property can be expressed for a fixed value, say 42, using the formula:

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\, b = 42).$$

In order to verify just this property, it is possible to use the following abstraction for a and b

$$h(i) = \begin{cases} 0, & \text{if } i = 42; \\ 1, & \text{otherwise.} \end{cases}$$

When this abstraction is applied and the program is compiled, the transition relation $\widehat{R}(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}')$ defined by $\widehat{b}' = \widehat{a}$ is obtained. Here, the primes denote next-state variables, and all of the variables range over $\{0, 1\}$. Now to check that our program works correctly for the value 42, the following formula will be checked at the abstract level:

$$\mathbf{AG}(\widehat{a} = 0 \rightarrow \mathbf{AX}\, \widehat{b} = 0).$$

The formula would of course turn out to be satisfied. Obviously, though, we do not want to have to repeat this process for each possible data value.

Suppose now that our abstraction function is modified as follows:

$$h_c(i) = \begin{cases} 0, & \text{if } i = c; \\ 1, & \text{otherwise.} \end{cases}$$

The resulting abstraction depends on a new symbolic parameter. Imagine compiling the program with this abstraction; a relation $\widehat{R}_c(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}', c)$ that is parameterized by c is

obtained. Fixing $c = 42$ will give the relation $\widehat{R}$ encountered above. If it were possible to run the model-checking algorithm on our parameterized relation, we would obtain a parameterized state set representing the states for which our formula is true. Now our specification

$$\mathbf{AG}(\widehat{a} = 0 \rightarrow \mathbf{AX}\,\widehat{b} = 0)$$

is essentially saying

$$\mathbf{AG}(a = c \rightarrow \mathbf{AX}\,b = c).$$

If the formula turns out to be true for all values of c, we will have proved the desired specification. The observation now is that by introducing eight extra OBDD variables to encode the possible choices for c, we can in fact:

1. represent h_c with an OBDD (the user will supply just h_c);
2. compile with h_c to get an OBDD representing $\widehat{R}_c(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}', c)$ (the compiler handles this step automatically);
3. perform the model checking to obtain an OBDD representing the parameterized state set (the model checker does this automatically; it simply views c as an additional state component that never changes); and
4. if necessary, choose a specific c and generate a counterexample (also done by the model checker).

Further note that, in this case, the program behaves identically regardless of the value of c, so when we compile it, the OBDD for $\widehat{R}_c$ will be independent of the extra variables that we introduced. As a result, doing the model checking will be no more complex than in the case when we were just verifying

$$\mathbf{AG}(a = 42 \rightarrow \mathbf{AX}\,b = 42).$$

In general, we have found that sharing in the OBDDs makes it possible to perform the abstraction, compilation, and model checking efficiently. We call abstractions such as h_c "symbolic abstractions."

We used symbolic abstractions to verify a simple pipeline circuit. This circuit is shown in Figure 13.6 and was described in detail in Section 6.5. It performs three-address arithmetic and logical operations on operands stored in a register file.

We used two independent abstractions to perform the verification. First, the register addresses were abstracted so that each address was either one of three symbolic constants (*ra*, *rb*, or *rc*) or some other value. This abstraction made it possible to collapse the entire register file down to only three registers, one for each constant. The second abstraction

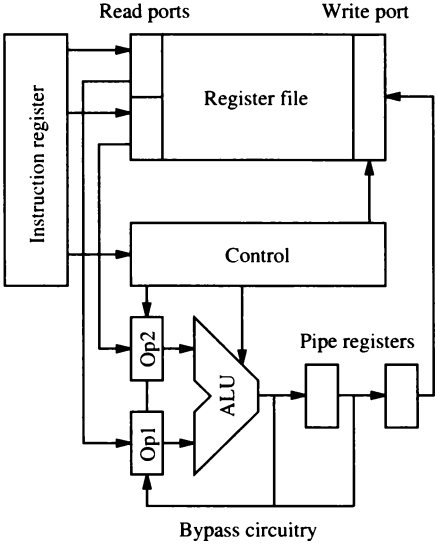


Figure 13.6
Pipeline circuit block diagram.

involved the individual registers in the system. In order to verify an operation, say addition, we create symbolic constants *ca* and *cb* and allow each register to be either *ca*, *cb*, *ca* + *cb* or some other value. As part of the specification, we verified that the circuit's addition operation works correctly. This property is expressed by the temporal formula

$$\mathbf{AG}\big((srcaddr1 = ra) \wedge (srcaddr2 = rb) \wedge (destaddr = rc) \wedge \neg stall$$
$$\rightarrow \mathbf{AX}\,\mathbf{AX}((regra = ca) \wedge (regrb = cb) \rightarrow \mathbf{AX}(regrc = ca + cb))\big).$$

This formula states that if the source address registers are *ra* and *rb*, the destination address register is *rc*, and the pipeline is not stalled, then the values in registers *ra* and *rb* two cycles from now will sum to the value in register *rc* three cycles from now. The reason for using the values of registers *ra* and *rb* two cycles in the future is to account for the latency in the pipeline.

The largest pipeline example we tried had sixty-four registers in the register file, and each register was 64 bits wide. This circuit has more than 4,000 state bits and nearly 10^{1300} reachable states. The verification required slightly less than six and a half hours of CPU time. In addition the verification times scale linearly in both the number of registers and

the width of the registers. For comparison, the largest circuit verified without abstraction had 8 registers, each 32 bits, and the verification required about four and a half hours of CPU time on a Sun 4. In addition the verification times there grew quadratically in the register width and cubicly in the number of registers. Techniques combining compositional reasoning and abstraction can be used to obtain even better results. These techniques are discussed in more detail elsewhere [179].

14 Symmetry

Finite state concurrent systems frequently exhibit considerable symmetry. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols—anything that has a lot of replicated structure. Recently, the use of symmetry in model checking has been investigated by several authors [58, 111, 143, 148]. These reduction techniques are based on the observation that having symmetry in the system implies the existence of nontrivial permutation groups that preserve both the state labeling and the transition relation. Such groups can be used to define an equivalence relation on the state space of the system. The quotient model induced by this relation is often smaller than the original model. Moreover, it is bisimulation equivalent to that model. Thus, it can be used to verify any property of the original model expressed by a CTL* formula.

14.1 Groups and Symmetry

We start by introducing some notions of group theory. Let G be a set. A *group* is a set G together with a binary operation on G, called the *group multiplication*, such that

- Multiplication is associative, that is, $a \circ (b \circ c) = (a \circ b) \circ c$.
- There is an element $e \in G$, called the *identity*, such that for all elements $a \in G$, $e \circ a = a = a \circ e$.
- For each element $a \in G$ there is an element a^{-1}, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

We usually use G to denote the group and concatenation to denote the multiplication operator. H is a *subgroup* of G if $H \subseteq G$ and H is a group under the multiplication operation of G.

Let G be a group and let $g_1, \ldots, g_k$ be designated elements of G. Define $\langle g_1, \ldots, g_k \rangle$ to be the smallest subgroup of G containing $g_1, \ldots, g_k$. If $H = \langle g_1, \ldots, g_k \rangle$, then we say that the group H is *generated* by the set $\{g_1, \ldots, g_k\}$. Note that H is the *closure* of the set $\{g_1, \ldots, g_k\}$ under the multiplication and inverse operations of G.

A *permutation* σ on a finite set of objects A is a bijection (i.e., a function that is one-to-one and onto) $\sigma : A \longrightarrow A$. The set of all permutations on A, denoted by $Sym\ A$, forms a group under functional composition. To see this, note that the identity permutation e is in $Sym\ A$; if $\sigma \in Sym\ A$, then its inverse, σ^{-1}, is in $Sym\ A$; and if $\sigma', \sigma'' \in Sym\ A$, then $\sigma = \sigma'' \circ \sigma' \in Sym\ A$. (In the expression $\sigma'' \circ \sigma'$ we apply σ' first and then apply σ''.) $Sym\ A$ is called the *full symmetric group*. A subgroup G of $Sym\ A$ is called a *permutation group* on A.

Two permutations σ_1, σ_2 are *disjoint* iff

$$\{i \mid \sigma_1(i) \neq i\} \cap \{j \mid \sigma_2(j) \neq j\} = \emptyset$$

A permutation that maps

$$i_1 \mapsto i_2,\ i_2 \mapsto i_3,\ \ldots,\ i_{k-1} \mapsto i_k,\ i_k \mapsto i_1$$

is called a *cycle* and is denoted by $(i_1\ i_2\ \ldots\ i_k)$. A cycle of length two is called *transposition*. It is possible to show that every finite permutation can be written as a composition of disjoint cycles, Moreover, every permutation can be written as a composition of transpositions, that are not necessarily disjoint [182].

For example, consider the permutation σ on $A = \{1, 2, 3, 4, 5\}$ given by

$$1 \mapsto 3,\ 2 \mapsto 4,\ 3 \mapsto 1,\ 4 \mapsto 5,\ 5 \mapsto 2.$$

σ can be written as a composition of disjoint cycles by $(1\ 3) \circ (2\ 4\ 5)$ and also as a composition of transpositions $(1\ 3) \circ (2\ 5) \circ (2\ 4)$. The subgroup of $Sym\ A$ generated by the two permutations $(1\ 3)$ and $(2\ 4\ 5)$ is a set with 6 elements:

$$\{e,\ (1\ 3),\ (2\ 4\ 5),\ (2\ 5\ 4),\ (1\ 3)(2\ 4\ 5),\ (1\ 3)(2\ 5\ 4)\ \}.$$

Let $M = (S, R, L)$ be a Kripke structure. Let G be a permutation group on the state space S of the structure M. A permutation $\sigma \in G$ is said to be a *automorphism* of M if and only if it preserves the transition relation R. More formally, σ should satisfy the following condition:

$$\forall s_1 \in S,\ \forall s_2 \in S,\ ((s_1, s_2) \in R \Rightarrow (\sigma(s_1), \sigma(s_2)) \in R)\ .$$

G is an *automorphism group* for the Kripke structure M if and only if every permutation $\sigma \in G$ is an *automorphism* of M. Notice that our definition of an automorphism group does not refer to the labeling function L. Further, note that because every $\sigma \in G$ has an inverse, which is also an automorphism, it can be proved that a permutation $\sigma \in G$ is an automorphism for a Kripke structure if and only if σ satisfies the following condition:

$$(\forall s_1 \in S)(\forall s_2 \in S)\ ((s_1, s_2) \in R \Leftrightarrow (\sigma(s_1), \sigma(s_2)) \in R)$$

It is easy to see that if every generator of the group G is an automorphism of M, then the group G is an automorphism group for M.

As an example, consider a simple token ring algorithm with one component process Q and many component processes P. Both P and Q have the structure shown in Figure 14.1. Each component process has three states: n (noncritical section), t (has the token), and c (critical section). There are two visible actions in the process: s (send token), and r (receive token). We also have a silent, internal action denoted by τ. For simplicity, this action is not shown in the figures. Process Q is initially in the state t, and process P is initially in the

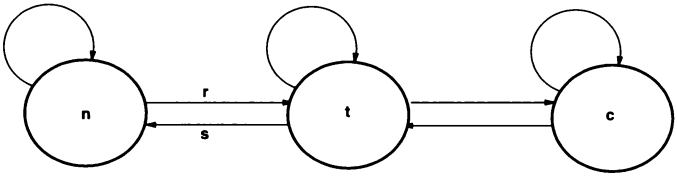


Figure 14.1
A process component.

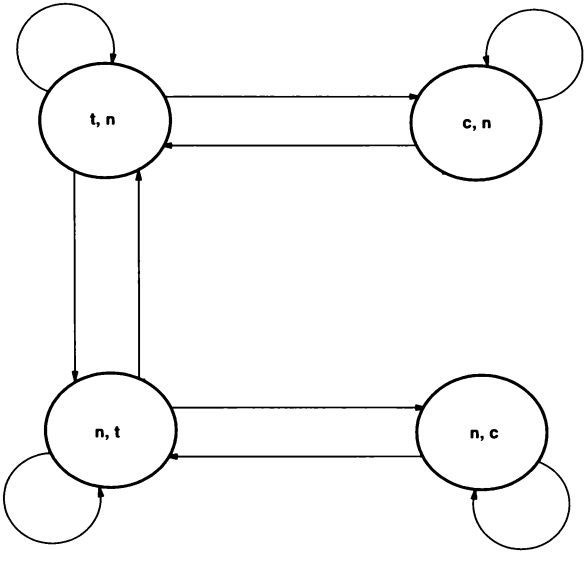


Figure 14.2
The Kripke structure for $Q \| P$.

state n. Composition of processes is synchronous. In the composition $Q \| P$, P and Q can either synchronize on the s action of Q and the r action of P or they can synchronize on the r action of Q and the s action of P. In both cases this results in an internal action τ. In addition, they can each perform an internal action τ. The Kripke structure corresponding to $Q \| P$ is shown in Figure 14.2. Let P^i be the composition of the process P, i times. In the token ring $Q \| P^i$, the s action of each process is synchronized with the r action of its right neighbor and its r action is synchronized with the s action of its left neighbor.

Let σ be a permutation acting on the state set of $Q \| P$, which exchanges (n, t) with (t, n) and (n, c) with (c, n). To see that σ is an automorphism for $Q \| P$, we examine the transition

from (t, n) to (c, n) and observe that there is also a transition from $\sigma((t, n)) = (n, t)$ to $\sigma((c, n)) = (n, c)$. Every other transition of $Q \| P$ is examined in a similar manner. Because each one of the transitions is preserved by σ, σ is an automorphism of $Q \| P$.

More generally, the behavior of a finite-state system is frequently determined by the values of a set of state variables $x_1, x_2, \ldots, x_n$ whose values are taken from some finite data domain D. For instance, a state of $Q \| P^i$ is an $i + 1$-tuple of state variables, each of which ranges over the data domain $\{n, t, c\}$.

When we extract a Kripke structure from a system, the values of the state variables determine the atomic propositions. The resulting Kripke model $M = (S, R, L)$ will have the following components:

- $S \subseteq D^n$, where each state can be thought of as an assignment of values to the n state variables.
- $R \subseteq S \times S$, where R is determined by the behavior of the system.
- The labeling function L is defined so that $d_i \in L(s)$ if and only if $x_i = d$.

It is often the case that the automorphism group is given as a group acting on the indices of the state variables. For example, the permutation σ, defined on the state set of $Q \| P$ may be also described by the transposition (1 2), which switches the state components corresponding to the first and the second processes.

A permutation σ, acting on the set of indices $\{1, 2, \ldots, n\}$, defines a new permutation σ', acting on states in D^n, in the following manner:

$$\sigma'((x_1, x_2, \ldots, x_n)) = (x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$$

Given two states x and y in D^n, it is easy to see that $x \neq y$ implies $\sigma'(x) \neq \sigma'(y)$. Thus, σ' is a permutation on D^n. It is easy to show that a group G acting on the set $\{1, 2, \ldots, n\}$ induces a permutation group G_1 acting on the set D^n. Consequently, an automorphism on the structure of a circuit induces an automorphism on the state space of the circuit.

14.2 Quotient Models

Let G be a permutation group acting on the set S and let s be an element of S, then the *orbit* of s is the set $\theta(s) = \{ t \mid \exists \sigma \in G(\sigma(s) = t) \}$. From each orbit $\theta(s)$ we pick a representative which we call $rep(\theta(s))$. Intuitively, the quotient model is obtained by collapsing all the states in one orbit to a single representative state.

Formally, let $M = (S, R, L)$ be a Kripke structure and let G be an automorphism group acting on S. The *quotient structure* $M_G = (S_G, R_G, L_G)$ is defined as follows:

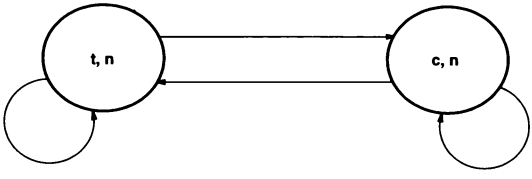


Figure 14.3
The quotient model for $Q \| P$.

- The state set is $S_G = \{ \theta(s) \mid s \in S \}$, the set of orbits of the states in S;
- The transition relation R_G is given by

$$R_G = \{ (\theta(s_1), \theta(s_2)) \mid (s_1, s_2) \in R \}; \tag{14.1}$$

- The labeling function L_G is given by $L_G(\theta(s)) = L(rep(\theta(s)))$.

Note that because G is an automorphism group, R_G is well defined and is independent of the chosen representatives. The definition of L_G, on the other hand, is not independent of the chosen representatives. To avoid this problem, we restrict our attention to symmetry groups that are also *invariance groups*.

G is an invariance group for an atomic proposition p if and only if the set of states labeled by p is closed under the application of all the permutations of G. More formally, an automorphism group G of a Kripke structure $M = (S, R, L)$ is an invariance group for an atomic proposition p if and only if the following condition holds:

$$(\forall \sigma \in G)(\forall s \in S)(p \in L(s) \Leftrightarrow p \in L(\sigma(s)))$$

We then say that p is an *invariant* under G. The notions of invariance group and invariant are extended to boolean formulas in a straightforward way.

To illustrate some of the notions defined above, consider again the Kripke structure $Q \| P$ in Figure 14.2. Let $G = \langle (1\ 2) \rangle$ be the group generated by $(1\ 2)$. Note that G is an automorphism group of $Q \| P$. In order to define the quotient model of $Q \| P$, induced by G, we first note that the orbits induced by G are:

$$\{(t, n), (n, t)\} \quad \text{and} \quad \{(c, n), (n, c)\}$$

If we pick the states (t, n) and (c, n) as representatives, the resulting quotient model is shown in Figure 14.3.

The Kripke structure corresponding to $Q \| P^i$ has $2(i + 1)$ reachable states. The permutation group $G = \langle (1\ 2\ \ldots\ i + 1) \rangle$ is an automorphism group for $Q \| P^i$. As in the case of $Q \| P$, G induces only two orbits,

$$\{(t, n^i), (n, t, n^{i-1}), \ldots, (n^i, t)\} \quad \text{and} \quad \{(c, n^i), (n, c, n^{i-1}), \ldots, (n^i, c)\}.$$

Thus, the quotient model for $Q \| P^i$ is identical to that of $Q \| P$, as shown in Figure 14.3. This example clearly demonstrates how exploiting symmetry can result in considerable savings.

Let c_i denote the boolean variable c for the i-th component, that is, if c_i is true, then the i-th process is in the critical section. Observe that G is an invariance group for the boolean formula me (for mutual exclusion) defined below:

$$me = (c_1 \rightarrow \neg c_2) \wedge (c_2 \rightarrow \neg c_1)$$

The theorem below states that if a temporal specification f has only invariant propositions, then f can be safely checked in the quotient model. We first present the following lemma, needed for the proof of the theorem.

LEMMA 39 Let $M = (S, R, L)$ be a Kripke structure with AP as the set of atomic propositions, let G be an invariance group for all propositions in AP, and let M_G be the quotient model for M. Moreover, let $B \subseteq S \times S_G$ be a relation defined by

$$\text{for every } s \in S, \quad B(s, \theta(s)).$$

Then, B is a bisimulation relation between M and M_G.

Proof To prove that B is a bisimulation, we first show that $L(s) = L_G(\theta(s))$. By the definition of M_G, we have $L_G(\theta(s)) = L(rep(\theta(s)))$. Since $rep(\theta(s)) \in \theta(s)$, there must be a permutation $\sigma \in G$ such that $\sigma(s) = rep(\theta(s))$. Because G is an invariance group for all $p \in AP$, we have that

$$\text{for all } p \in AP, \ p \in L(rep(\theta(s))) \Leftrightarrow p \in L(s).$$

Thus, $L(s) = L(rep(\theta(s))) = L_G(\theta(s))$.

Consider a transition $(s, t) \in R$. Then, by the definition of R_G, $(\theta(s), \theta(t)) \in R_G$. Moreover by the definition of the relation B we have that $B(t, \theta(t))$.

Now let ϑ be a state in S_G such that $(\theta(s), \vartheta) \in R_G$. ϑ contains at least one element, namely $rep(\vartheta)$. Let t be equal to $rep(\vartheta)$. Then, $\vartheta = \theta(t)$ and $(\theta(s), \vartheta) \in R_G$ can be rewritten as $(\theta(s), \theta(t)) \in R_G$. By the definition of R_G, this means there exist two states s_1 and t_1 such that $(s_1, t_1) \in R$, $s_1 \in \theta(s)$, and $t_1 \in \theta(t)$. Because s_1 and s belong to the same orbit, there exists a permutation $\sigma_1 \in G$ such that $\sigma(s_1) = s$. By definition of a symmetry group, $(\sigma(s_1), \sigma(t)) \in R$, or in other words $(s, \sigma(t)) \in R$. Notice that t and $\sigma(t)$ belong in the same orbit. Hence, $\sigma(t) \in \vartheta$ and by definition of B, $B(\sigma(t), \vartheta)$. □

By Theorem 13, the previous lemma immediately implies the following corollary.

COROLLARY 4 Let M be a structure defined over AP and let G be an invariance group for AP. Then, for every $s \in S$ and every CTL* formula defined over AP,

$$M, s \models f \Leftrightarrow M_G, \theta(s) \models f.$$

THEOREM 28 Let $M = (S, R, L)$ be a Kripke structure, G be an automorphism group of M, and f be a CTL* formula. If G is an invariance group for all the atomic propositions p occurring in f, then

$$M, s \models f \Leftrightarrow M_G, \theta(s) \models f \tag{14.2}$$

where M_G is the quotient structure corresponding to M.

Proof Assume that M is defined over AP and f is defined over $AP' \subseteq AP$. The *restriction* of M to AP' is the structure $M' = (S, R, L')$ that is identical to M, except that for every $s \in S$, $L'(s) = L(s) \cap AP'$. Clearly, for every CTL* formula defined over AP' and for every $s \in S$,

$$M, s \models f \Leftrightarrow M', s \models f.$$

Let M'_G be the quotient model of M', induced by G. By the definition of quotient model, M'_G is the restriction of M_G to AP'. Thus, for every $\vartheta \in S_G$,

$$M_G, \vartheta \models f \Leftrightarrow M'_G, \vartheta \models f.$$

Because G is an invariance group for AP', corollary 4 applies and we have:

$$M', s \models f \Leftrightarrow M'_G, \theta(s) \models f.$$

Altogether, we conclude that

$$M, s \models f \Leftrightarrow M_G, \theta(s) \models f. \quad \square$$

14.3 Model Checking with Symmetry

In this section, we describe how to perform model checking in the presence of symmetry. First, we discuss how to find the set of states in a Kripke structure that are reachable from a given set of initial states using an explicit state representation. In the explicit state case, a breadth-first or depth-first search starting from the set of initial states is performed. Typically, two lists, a list of reached states and a list of unexplored states, are maintained. At the beginning of the algorithm, the initial states are put on both the lists. In the exploration step, a state is removed from the list of unexplored states and all its successors are processed. An algorithm for exploring the state space of a Kripke structure

```
reached := ∅;
unexplored := ∅;
for all initial states s do
    append ξ(s) to reach;
    append ξ(s) to unexplored;
end for all
while unexplored ≠ ∅ do
    remove a state s from unexplored;
    for all successor states q of s do
        if ξ(q) is not in reached
            append ξ(q) to reached;
            append ξ(q) to unexplored;
        end if
    end for all
end while
```

Figure 14.4
Exploring state space in presence of symmetry.

in the presence of symmetry is discussed in [148]. The authors introduce a function $\xi(q)$, which maps a state q to the unique state representing the orbit of that state. While exploring the state space, only the unique representatives from the orbits are put on the list of reached and unexplored states. An outline of the algorithm is shown in Figure 14.4. This simple reachability algorithm can be extended to a full CTL model checking algorithm by using the technique described in [63]. In order to construct the function $\xi(q)$ it is important to compute the orbit relation efficiently.

When OBDDs are used as the underlying representation, the construction of the quotient model is more complex. First note that if R is represented by the OBDD $R(v_1, \ldots, v_k, v'_1, \ldots, v'_k)$ and σ is a permutation on the state variables, it is straightforward to check that σ is an automorphism of M. This is done by checking that $R(v_1, \ldots, v_k, v'_1, \ldots, v'_k)$ is identical to $R(v_{\sigma(1)}, \ldots, v_{\sigma(k)}, v'_{\sigma(1)}, \ldots, v'_{\sigma(k)})$, which is the OBDD representing the transition relation of the permuted structure.

Our method of computing the quotient model uses the OBDD for the *orbit relation* $\Theta(x, y) \Leftrightarrow (x \in \theta(y))$. Given a Kripke structure $M = (S, R, L)$ and an automorphism group G on M with r generators $g_1, g_2, \ldots, g_r$, the orbit relation Θ is the least fixpoint of the equation given below:

$$Y(x, y) = (x = y \vee \exists z\, (Y(x, z) \wedge \bigvee_i y = g_i(z))) \tag{14.3}$$

This result is proved in the next lemma.

LEMMA 40 The least fixpoint of Equation 14.3 is the orbit relation Θ induced by the group G generated by $g_1, g_2, \ldots, g_r$.

Proof First, we prove that Θ is a fixpoint of Equation 14.3. It is obvious by the transitivity and reflexivity of the orbit relation Θ that

$$\Theta(x, y) \supseteq (x = y \vee \exists z\ (\Theta(x, z) \wedge \bigvee_i y = g_i(z))).$$

Suppose $\Theta(x, y)$, then $\Theta(y, x)$ holds as well. Thus, by the definition of the orbit relation there exists $\sigma \in G$ such that $y = \sigma(x)$. Let us assume $x \neq y$ (if $x = y$, the result is immediate). This means there exists a generator g_k, $k \leq r$ such that $y = g_k(\sigma_1(x))$. Setting $z = \sigma_1(x)$, we see that $\Theta(x, z)$ and $y = g_k(z)$. Because x and y are arbitrary boolean vectors, we get the following inclusion:

$$\Theta(x, y) \subseteq (x = y \vee \exists z\ (\Theta(x, z) \wedge \bigvee_i y = g_i(z)))$$

Hence Θ is a fixpoint of Equation 14.3.

Next, we prove that if T is any fixpoint of equation 14.3, then $\Theta \subseteq T$. We prove that $\Theta(x, y) \Rightarrow T(x, y)$. The definition of the orbit relation $\Theta(x, y)$ implies that there exists a $\sigma = g_{i_m} \ldots g_{i_2} g_{i_1}$, $1 \leq i_j \leq r$ such that $\sigma(x) = y$. Because T is a fixpoint of Equation 14.3, it can be proved by induction that for all $1 \leq l \leq m$, $T(x, g_{i_l} \ldots g_{i_1}(x))$ holds. Using this result for $l = m$, we see that $T(x, y)$ holds. Since $\Theta(x, y) \Rightarrow T(x, y)$, we obtain that $\Theta \subseteq T$. Hence, Θ is the least fixpoint. □

If a suitable state encoding is available, this fixpoint equation can be computed using OBDDs [46]. Once we have the orbit relation Θ, we need to compute a function $\xi : S \rightarrow S$, which maps each state s to the unique representative in its orbit. If we view states as vectors of values associated with the state variables, it is possible to choose the lexicographically smallest state to be the unique representative of the orbit. Because Θ is an equivalence relation, these unique representatives can be computed using OBDDs by the method of Lin and Newton [175].

Assuming that we have the OBDD representation of the mapping function ξ, the transition relation R_G of the quotient structure can be expressed as follows:

$$R_G(x, y) = \exists x_1 \exists y_1 (R(x_1, y_1) \wedge \xi(x_1) = x \wedge \xi(y_1) = y)$$

14.4 Complexity Issues

In this section we consider complexity issues that arise in exploiting symmetry for model checking. We show that the orbit problem is at least as hard as the graph isomorphism problem, which is in NP, but not known to be NP complete. We also prove bounds on the size of the OBDD for the orbit relation.

14.4.1 The Orbit Problem and Graph Isomorphism

The most basic step in performing model checking with symmetry is to decide whether two states are in the same orbit. We now discuss the complexity of this problem.

Let G be a group acting on the set $\{1, 2, \ldots, n\}$. Assume that G is represented in terms of a finite set of generators. Given two vectors $x \in B^n$ and $y \in B^n$, the *orbit problem* asks whether there exists a permutation $\sigma \in G$ such that $y = \sigma(x)$.

Given two graphs $\Gamma_1 = (V_1, E_1)$ and $\Gamma_2 = (V_2, E_2)$ such that $|V_1| = |V_2|$, the *Graph Isomorphism problem* asks whether there exists a bijection $f : V_1 \to V_2$ such that the following condition holds

$$(i, j) \in E_1 \Leftrightarrow (f(i), f(j)) \in E_2$$

THEOREM 29 The orbit problem is as hard as the Graph Isomorphism problem.

Proof Given two graphs $\Gamma_1 = (V_1, E_1)$ and $\Gamma_2 = (V_2, E_2)$ we construct a group G and two $0-1$ vectors x and y such that x and y are in the same orbit under the action of the group G if and only if Γ_1 and Γ_2 are isomorphic. We assume that $|V_1| = |V_2| = n$. Let $A = \{a_{ij}\}$ and $B = \{b_{ij}\}$ be the adjacency matrices of the graph Γ_1 and Γ_2 respectively. Let $x \in \{0, 1\}^{n^2}$ be defined as follows:

$$x_{n(i-1)+j} = a_{ij}, \quad 1 \le i \le n, \quad 1 \le j \le n$$

The vector $x \in \{0, 1\}^{n^2}$ is a list of the elements of the matrix A in row order. The vector $y \in \{0, 1\}^{n^2}$ is defined in a similar fashion using the adjacency matrix B. Let $(i\ j)$ be a transposition acting on the set $\{1, 2, \ldots, n\}$. Intuitively, we can think of this transposition as exchanging the vertices i and j in the graph Γ_1. This corresponds to exchanging the rows i and j and columns i and j in the adjacency matrix and has exactly the same effect as applying the permutation σ given below to the vector x.

$$\sigma_{row} = (n(i-1)+1, n(j-1)+1) \ldots (n(i-1)+n, n(j-1)+n)$$
$$\sigma_{col} = (i, j) \ldots ((n-1)n+i, (n-1)n+j)$$
$$\sigma = \sigma_{row}\sigma_{col}$$

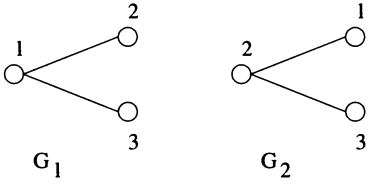


Figure 14.5
Two isomorphic graphs.

Each permutation acting on the set of size $n = |V_1|$ corresponds to a bijection $f : V_1 \mapsto V_2$. We assume that the vertices are labeled by integers. If the bijection corresponding to the permutation $(i\ \ j)$ is an isomorphism between Γ_1 and Γ_2, then exchanging rows i and j and columns i and j in the adjacency matrix A gives B. This implies that $y = \sigma(x)$ because x and y are just encodings of the adjacency matrix A and B respectively. Similarly, if $y = \sigma(x)$, then the bijection corresponding to the permutation $(i\ \ j)$ is an isomorphism between the graph Γ_1 and Γ_2. Therefore, $y = \sigma(x)$ if and only if the bijection corresponding to the permutation $(i\ \ j)$ is an isomorphism between Γ_1 and Γ_2. Every bijection $f : V_1 \mapsto V_2$ corresponds to some permutation in the full symmetric group S_n. Since the group S_n acting on the set $\{1, 2, \ldots, n\}$ is generated by the transpositions $(1\ \ 2)$,$(1\ \ 3)$, ... and $(1\ \ n)$, we have the result. We just have to code all these transpositions in the context of the $0-1$ vectors x and y. □

As an example, consider the two graphs Γ_1 and Γ_2 given in Figure 14.5. The vectors x and y given below encode the adjacency matrices of the graphs Γ_1 and Γ_2 respectively:

$$x = (011\ \ 100\ \ 100)$$

$$y = (010\ \ 101\ \ 010)$$

The permutations σ_{row} and σ_{col} below exchange rows 1 and 2 and column 1 and 2, respectively, in the matrix described by x. Their composition corresponds to exchanging vertices 1 and 2 in graph Γ_1.

$$\sigma_{row} = (1\ \ 4)(2\ \ 5)(3\ \ 6)$$

$$\sigma_{col} = (1\ \ 2)(4\ \ 5)(7\ \ 8)$$

$$\sigma = \sigma_{row} \circ \sigma_{col}$$

Notice that $y = \sigma(x)$ and the bijection corresponding to the permutation $(1\ \ 2)$ is an isomorphism between Γ_1 and Γ_2.

14.4.2 The Orbit Relation and OBDDs

Circuits are typically built from components and the state bits are grouped according to the hierarchical structure of the system. In practice two types of symmetry groups occur frequently:

- *rotation groups*, when equivalent components are ordered cyclically and can be rotated any number of steps. For example, the token ring protocol used in the solution to the distributed mutual exclusion problem exhibits rotational symmetry. A permutation group G acting on $\{1, 2, \ldots, n\}$ is a *rotation group* if it is generated by the cycle $(1\ 2\ \ldots\ n)$
- *full symmetric groups*, when equivalent components are unordered and can be exchanged arbitrarily. Such groups occur for example in systems where components communicate via a common bus (e.g., multiprocessor systems) or in systems where broadcasting is used.

We will only prove a lower bound on the size of the OBDD for the orbit relation of rotation groups. The proof for full symmetric groups is similar and is given in [58].

For simplicity we consider a system built by the composition of N instances of one component, for example, a ring or bus with N equivalent components. One component i is represented by a vector $\vec{x}_i$ of k state variables $x_{i,1}, \ldots, x_{i,k}$. We will refer to such a vector as a *block*. The state of the system is represented by $\langle \vec{x}_1, \ldots, \vec{x}_N \rangle$. A permutation σ acting on the components $\{1, \ldots, N\}$ induces a permutation on the state variables and hence also a permutation on the set of states: $\sigma(\langle \vec{x}_1, \ldots, \vec{x}_N \rangle) = \langle \vec{x}_{\sigma(1)}, \ldots, \vec{x}_{\sigma(N)} \rangle$.

The OBDD for the orbit relation Θ of a group G ranges over the variables $\vec{x}_1, \ldots, \vec{x}_N$, $\vec{x'}_1, \ldots, \vec{x'}_N$ and is defined by:

$$\Theta(\vec{x}_1, \ldots, \vec{x}_N, \vec{x'}_1, \ldots, \vec{x'}_N) = 1$$

if and only if

$$\exists \sigma \in G : \sigma(\langle \vec{x}_1, \ldots, \vec{x}_N \rangle) = \langle \vec{x'}_1, \ldots, \vec{x'}_N \rangle.$$

The size of the OBDD representing Θ is denoted by $|\Theta|$.

LEMMA 41 Let $f(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ be the following boolean function:

$$\bigwedge_{i=1}^{n} (x_i = x'_i)$$

Let F be the OBDD for f such that all the unprimed variables are ordered before all the primed variables. In this case $|F| \geq 2^n$.

Proof Consider two distinct assignments $(b_1, \ldots, b_n)$ and $(c_1, \ldots, c_n)$ to the boolean vector $(x_1, \ldots, x_n)$. These two assignments can be distinguished because of the following equation:

$$f(b_1, \ldots, b_n, b_1, \ldots, b_n) \neq f(c_1, \ldots, c_n, b_1, \ldots, b_n)$$

Let v_1 and v_2 be the nodes reached after following the path $(b_1, \ldots, b_n)$ and $(c_1, \ldots, c_n)$ from the top node. Because these two assignments can be distinguished, we must have $v_1 \neq v_2$. There are 2^n different assignments to the boolean vector $(x_1, \ldots, x_n)$ and each of them corresponds to a different node (at level n) in the OBDD F. Therefore, the number of nodes at level n in the OBDD F is $\geq 2^n$. □

THEOREM 30 Let the state of a system be composed of N equivalent components each with k state variables. For a rotation group G acting on the set $\{1, \ldots, N\}$ we have the following lower bound for the OBDD representing the induced orbit relation Θ.

$$|\Theta| > 2^K \text{ with } K = \min(\sqrt{N}, 2^{k-1})$$

Proof Let Θ be the OBDD for the orbit relation. For the proof we consider the first variable of each block. From the top of the OBDD Θ we go down until we have K variables $x_{i,1}$ or K variables $x'_{i,1}$. We will cut the OBDD Θ at this level. Without loss of generality we assume that we have K unprimed variables with indices $I = \{i_1, \ldots, i_K\}$ above the cut. Let J be the set of indices of primed variables of the form $x'_{j,1}$ above the cut. The set J must contain less than K elements.

Let T be the following set:

$$T = \{\, \sigma \in G \mid \sigma(I) \cap J \neq \emptyset \,\}$$

For each permutation $\sigma \in T$ there exists $i \in I$, $j \in J$ such that the permutation σ rotates the i-th block to the j-th block. Because σ is a rotation, knowing that it maps i to j determines it. The number of ways of choosing $i \in I$ and $j \in J$ is less than K^2. It follows from the definition of K that $K^2 \leq N$ and, therefore, $|T| < N$ and $G - T$ is nonempty.

Any rotation $r \in G - T$ has the property that $r(I) \cap J = \emptyset$. In other words, each such rotation maps an unprimed variable $x_{i,1}$ that occurs above the cut to a primed variable $x'_{j,1}$ that occurs below the cut.

Our goal is to use Lemma 41 to bound the size of the OBDD Θ. In order to accomplish this, we construct an OBDD Θ' that is smaller than Θ and has the property that all of the unprimed variables occur before all of the primed variables.

Choose a rotation $r \in G - T$. Instantiate the variables $\langle x_{i_j,2}, \ldots, x_{i_j,k} \rangle$ and $\langle x'_{i_j+r,2}, \ldots, x'_{i_j+r,k} \rangle$ for $i_j \in I$ with the binary encoding of the number j. (Since $1 \leq j \leq K$, we need $K \leq 2^{k-1}$). The variables $x_{i,j}$ and $x'_{i+r,j}$ are instantiated with 0 for $i \notin I$.

The resulting OBDD Θ' has free variables $x_{i,1}, x'_{i+r,1}$ $i \in I$, where all the unprimed variables are above the cut and all primed variables are below, and is smaller than the OBDD Θ. The instantiation was chosen in such a way that for the rotation r the primed and unprimed variables must be equal. Thus, Θ' is the OBDD for the following boolean formula

$$\bigwedge_{i \in I} (x_{i,1} = x'_{i+r,1}).$$

Because the variables $x_{i,1}$ are ordered before the variables $x'_{i+r,1}$, it follows from Lemma 41 that the size of the OBDD Θ' is greater than 2^K. Because the OBDD for Θ' is smaller than the OBDD for Θ, the desired result follows. □

The OBDD of the orbit relation induced by a full symmetric or rotation group on the components is exponential in the minimum of the number of components and the number of states in one component. Consequently, using the orbit relation to exploit symmetries of that kind in symbolic model checking is restricted to examples with a small number of components or where each component has only a few states. An approach that avoids the computation of the orbit relation is described in [64]. Given a Kripke structure $M = (S, R, L)$ and a set of representatives $Rep \subseteq S$, their approach builds a model M_{Rep} whose state set is Rep. The set Rep can have more than one state from each orbit. This approach does not need the OBDD for the orbit relation.

14.5 Empirical Results

To test these ideas consider a simple cache coherence protocol for a single-bus multiprocessor system based on the Futurebus+ IEEE standard [147]. A discussion of this protocol is contained in Chapter 8.2. The system has a bus over which the processors and the global memory communicate. Each processor contains a local cache that consists of a fixed number of cache lines (see Figure 14.6).

In each bus cycle the bus arbiter chooses one processor to be the master. The master processor selects a cache line address and a command it wants to put on the bus. The other processors and the memory respond to the bus command and change their local context. The reaction of the components is described in the protocol standard, which enforces the coherence of the cache lines among the different processors, that is, only valid data values are read by the processors and no writes are lost. For the verification task, the protocol is formalized, and cache coherence and other important system properties are expressed in temporal logic.

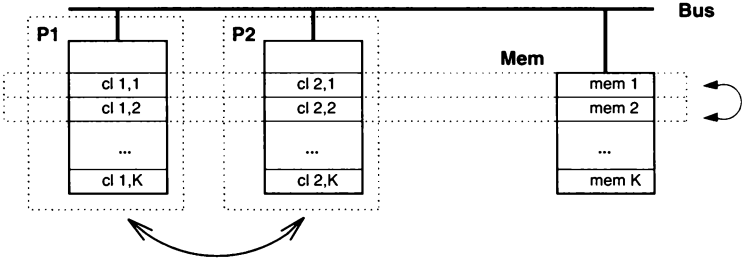


Figure 14.6
System structure.

The behavior of the processors, the bus, and the memory can be described by finite state machines. The state of the processor P_i is a combination of the states of each cache line in the processor cache and the state of the bus interface. The global bus is represented by the command on the bus, the active cache line address and other bus control signals, for example, for bus snooping and arbitration.

There are two obvious symmetries in the system. First, processors are symmetric, that is, we can exchange the context of any two processors in the system. Second, cache lines are symmetric, that is, any two cache lines can be exchanged simultaneously in all processors and the memory. To maintain consistency, along with applying the symmetries mentioned above all the cache lines and processor addresses in the system must be renamed. Both symmetries are indicated in Figure 14.6 by arrows.

The complete system is the synchronous composition of all the components and is described by a Kripke structure $M = (S, R, L)$. Because domains can be encoded in binary, a state is just a binary vector, and the transition relation R can be represented by an OBDD.

When we use only processor symmetry, we choose as the set of representatives the states where processor 1 is the master. When we use only cache symmetry, we choose as the set of representatives the states where cache line 1 is active. When we use both symmetries we choose the set of states where processor 1 is the master and cache line 1 is active as the set of representatives.

Consider the following properties, each of which can be represented by a propositional formula:

Property p For all cache lines it is true that if one processor is in *exclusive-modified* state, then all other processors are in the *invalid* state.

Property q For all cache lines it is the case that if memory has valid data, then either all processors are in *shared-unmodified* or *invalid* state, or one processor is in *exclusive-unmodified* state.

Property m All cache lines in memory are valid.

Property c The command on the bus is either *read-modified* or *invalidate*

Some important properties of the protocol are:

- **AG** p and **AG** q—the properties p and q always hold.
- **AG**$(m \rightarrow \mathbf{A}[m\ \mathbf{U}\ c])$—if the memory has valid data, then it remains valid until an appropriate command is issued.
- **AG**(**EF** m)—from all the reachable states it is possible to get to a state where the memory has valid data for all the cache lines,

In [64] symmetry is exploited in order to check these properties for a model of the cache consistency protocol, represented by OBDDs. For some configurations the OBDD sizes are reduced by a factor of 15.

15 Infinite Families of Finite-State Systems

The ability to reason automatically about entire families of finite-state systems is an important goal. Such families arise frequently in the design of reactive systems in both hardware and software. Typically, circuit and protocol designs are parameterized, that is, they define an infinite family of systems. For example, a circuit design to multiply two integers has the width of the integers n as a parameter; the design of a bus has the number of processors and caches on the bus parameterized, and in the design of a token-ring algorithm the number of processes on the ring is parameterized.

Most of the research done in the area of model checking focuses on verifying single finite-state systems. In this chapter we describe methods to verify parameterized designs, viewed as infinite families of finite-state systems. This problem can also be thought of as solving the state explosion problem because in this case the state set is unbounded. Formally, the problem can be stated as follows:

Given an infinite family of systems $\mathcal{F} = \{M_i\}_{i=1}^{\infty}$ and a
temporal formula f, verify that all the systems in $\mathcal{F}$ satisfy f,
that is, $\forall i\ [M_i \models f]$.

In general the problem is undecidable [12, 16, 237]. We give a formal proof of this result at the end of the chapter in Section 15.5. It is not necessary to understand the details of this proof in order to read the remainder of this chapter.

15.1 Temporal Logic for Infinite Families

Traditionally, temporal logics specify properties of a single Kripke structure. These logics can be extended to specify properties of infinite families of Kripke structures. Two such logics are discussed below.

In Browne, Clarke, and Grumberg [33], a version of CTL* called *indexed* CTL* or ICTL* is introduced. The propositions in ICTL* are indexed by the natural numbers. Intuitively, if a proposition is indexed by i, it applies to the i-th component process. Let f be an arbitrary CTL* formula. Let $f(i)$ be the formula f where all the propositions have been indexed by i. The indexed logic ICTL* permits formulas of the form $\wedge_i f(i)$ (the formula f is true in all components) and $\vee_i f(i)$ (the formula f is true in some component). One can also have formulas like $\wedge_{j \neq i} f(j)$ (every component but the i-th component satisfies f) or $\vee_{j \neq i} f(j)$ (some component other than the i-component satisfies f). For example, consider the infinite family of token rings $\mathcal{F} = \{Q \| P^i\}_{i=1}^{\infty}$. The ICTL* formula given below expresses the *mutual exclusion* property for the family $\mathcal{F}$.

$$\bigwedge_i \mathbf{AG}(c_i \Rightarrow \wedge_{i \neq j} \neg c_j)$$

In [68], another version of CTL* is proposed that replaces atomic propositions by regular expressions. Consider again the family $\mathcal{F} = \{Q \| P^i\}_{i=1}^{\infty}$ and let $S = \{n, t, c\}$. The states in any Kripke structure in $\mathcal{F}$ can be vectors of arbitrary size whose components are in S. In other words, the states of Kripke structures in $\mathcal{F}$ are strings over the alphabet S, and therefore belong to S^*. Notice that the regular expression $\{n, t\}^* c \{n, t\}^*$ represents the mutual exclusion property for a state in some structure in $\mathcal{F}$. The advantage of regular expressions is that they apply to arbitrary sized vectors over S and can characterize states in any Kripke structure belonging to the infinite family $\mathcal{F}$. The formula given below states the mutual exclusion property:

$\mathbf{AG}(\{n, t\}^* c \{n, t\}^*)$

15.2 Invariants

Most techniques for verifying families of finite-state structures rely on finding an *invariant*. Formally, an invariant can be defined as follows. Given a family $\mathcal{F} = \{M_1, M_2, \ldots\}$ and a reflexive, transitive relation $\geq$ on structures, an *invariant* $\mathcal{I}$ is a structure such that for all M in $\mathcal{F}, \mathcal{I} \geq M$.

The relation $\geq$ determines what kind of temporal property can be checked. The most widely used relations are the *bisimulation* equivalence ($M \equiv \mathcal{I}$) and the *simulation* preorder ($M \preceq \mathcal{I}$) that preserve the logics CTL* and ACTL*, and language equivalence ($M \cong \mathcal{I}$) and language inclusion ($M \subseteq \mathcal{I}$) that preserve the logic LTL. Both the bisimulation equivalence and the language equivalence provide *strong preservation*, that is, for all M in $\mathcal{F}$,

$\mathcal{I} \models f \Longrightarrow M \models f$, and

$\mathcal{I} \not\models f \Longrightarrow M \not\models f$.

The simulation preorder and language inclusion, on the other hand, provide only *weak preservation*, that is, for all M in $\mathcal{F}$.

$\mathcal{I} \models f \Longrightarrow M \models f$.

However, if $\mathcal{I} \not\models f$, then nothing can be concluded about the truth of f in the family and a new invariant has to be suggested. The counterexample generated while checking whether f is true in $\mathcal{I}$ can be a useful aid in guessing a new invariant.

In Browne, Clarke, and Grumberg [33], a family of token rings is considered. It is shown that a ring of size n ($n \geq 2$) is bisimilar to a ring of size 2. In this case the token ring of size 2 is the invariant $\mathcal{I}$. Let f be an arbitrary CTL* formula. Using Theorem 14, we have that $\mathcal{I} \models f$ iff f is true in the entire family of token rings. Unfortunately, the bisimulation

has to be constructed manually. Moreover, because the bisimulation equivalence $\equiv$ is more stringent than the simulation preorder $\preceq$, it is harder to devise an invariant for $\equiv$. Alternative techniques for reasoning about families of finite state structures are given in [109, 110, 123].

McMillan and Kurshan [165] and Wolper and Lovinfosse [250] suggest an *invariant rule* as a more systematic way for establishing an invariant. Assume that each member M_i in the family $\mathcal{F}$, is a composition of some number of basic structures. Further assume that the composition operator $\|$ is *monotonic* with respect to the relation $\geq$, that is, for all structures P_1, P_1', P_2, P_2', if $P_1 \geq P_1'$ and $P_2 \geq P_2'$ then $P_1 \| P_2 \geq P_1' \| P_2'$.

The *invariant rule* in its simplest form is given for the family $\mathcal{F} = \{P^i\}_{i=1}^{\infty}$. The lemma below states the invariant rule and proves its correctness.

LEMMA 42 Let $\geq$ be a reflexive, transitive relation and let $\|$ be a composition operator that is monotonic with respect to $\geq$. If $\mathcal{I} \geq P$ and $\mathcal{I} \geq \mathcal{I} \| P$, then $\mathcal{I} \geq P^i$, for all $i \geq 1$.

Proof We prove the result by induction on i. Using the hypothesis, the result is true for $i = 1$. Let $i \geq 2$ and assume that the result is true for $i - 1$. The first equation given below is the induction hypothesis. The second equation follows from the first by composing with process P and using the monotonicity of composition with respect to $\geq$.

$$\mathcal{I} \geq P^{i-1}$$

$$\mathcal{I} \| P \geq P^i$$

Now using the fact that $\mathcal{I} \geq \mathcal{I} \| P$ and the transitivity of $\geq$, we get that $\mathcal{I} \geq P^i$. □

This rule can easily be extended to families of the form $\{Q \| P^i\}_{i=1}^{\infty}$ for any structures Q and P. If $\mathcal{I}'$ satisfies the conditions above, then $\mathcal{I} = Q \| \mathcal{I}'$ is an invariant for this family. This rule is still valid if other operations on processes (like renaming and hiding) are allowed, provided that they are monotonic with respect to $\geq$.

Sometimes it is difficult or even impossible to find an invariant $\mathcal{I}$ such that $\mathcal{I} \geq P^i$. However, if we consider the environment in which the P^is are running (for instance Q in the above example), then such an invariant exists. We show how this technique can be used to verify the token ring example given in Chapter 14. The processes P and Q from this example are reproduced in Figure 15.1. They are identical except that the initial state of Q is t whereas the initial state of P is n. Figure 15.2 gives the structure $Q \| P$ corresponding to the composition of Q and P. The composition operator $\|$ is defined in a similar manner to the composition operator in Chapter 12 and is monotonic with respect to the simulation preorder.

We claim that $Q \| P$ is an invariant for the family $\{Q \| P^i\}_{i=1}^{\infty}$ with respect to the simulation preorder ($\succeq$). To prove this, we need to show only that $Q \| P \succeq Q \| P \| P$.

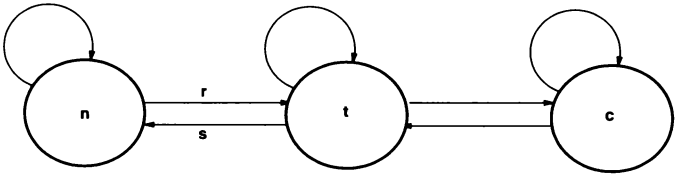


Figure 15.1
A process component.

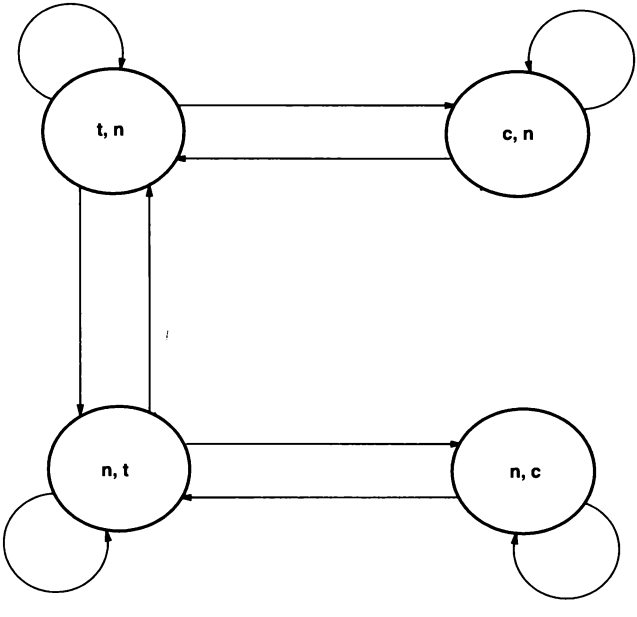


Figure 15.2
The Kripke structure for $Q \| P$.

By monotonicity of $\|$ with respect to $\succeq$ and the transitivity of $\succeq$ we conclude that $Q \| P \succeq Q \| P^i$, for every i.

$Q \| P \| P$ is given in Figure 15.3. The simulation relation associates the initial state (t, n, n) of $Q \| P \| P$ with the initial state (t, n) of $Q \| P$. It also associates (c, n, n) with (c, n). States (n, t, n) and (n, n, t) are associated with (n, t), and states (n, c, n) and (n, n, c) are associated with (n, c). It is easy to check that this relation is a simulation preorder.

In [165] and [250], extensions of the invariant rule are applied in the context of specific models of computation with specific preorders.

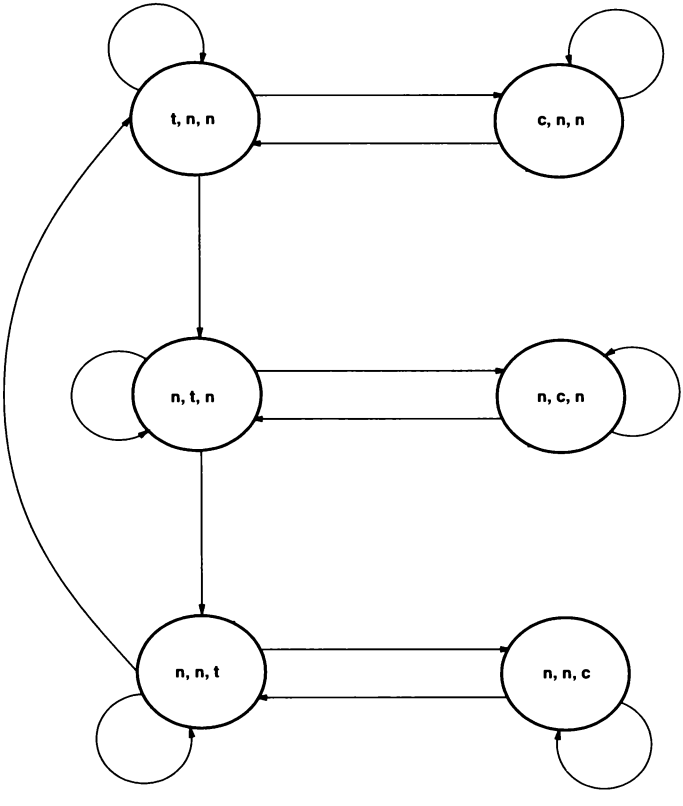


Figure 15.3
The Kripke structure for $Q \| P \| P$.

15.3 Futurebus+ Example Reconsidered

In this section we apply the induction principle to a nontrivial example. We consider the Futurebus+ cache-coherence protocol discussed in Chapters 8.2 and 14.5 for the case of a single bus. This example is described by the infinite family of Kripke structures $\mathcal{F} = \{P^1, P^2, \ldots\}$, where P^i represents a bus with i processes on it. Each component structure P is given by an SMV program. The portion of the program describing how the next command is generated for process P is given in Figure 15.4. We abbreviate the state values to be I for *invalid*, EM for *exclusive – modified*, EU for *exclusive – unmodified*, and SU for *shared – unmodified*. Each processor has a boolean variable `master` that is true when the processor has write permission to the bus. Exactly one processor has its `master` variable set to 1 at any time.

```
ASSIGN
init(cmd) := idle;
next(cmd) :=
 case
  state = I & !master : {read_shared, read_modified, idle};
  state = EM & !master : {copy_back, idle};
  state = EU & !master : {copy_back, idle};
  state = SU & !master : {invalidate, copy_back, idle};
  master : cmd;
  1 : idle;
 esac;
```

Figure 15.4
Command part for the process P.

Our first approximation for the invariant is the process P. By the induction principle, for P to be an invariant it should satisfy

$$P \succeq P \| P$$

that is, P should "mimic" the behavior of $P \| P$. Unfortunately, this does not hold. For instance, when $P \| P$ is in the state (*exclusive-modified*, *invalid*), it can issue the commands *copy-back* and *read-shared*. No state in P can issue both of these commands. To solve this problem, we guess a modification, called P', as the new invariant. P and P' differ mainly in the way they issue the next command. The portion of the modified program for P' is given in Figure 15.5.

To prove that P' is an invariant we have to check the following conditions:

$$P' \succeq P$$

$$P' \succeq P' \| P$$

The first requirement holds because P' is derived from P by adding more transitions. To prove that the second requirement holds, we establish a correspondence between reachable states in P' and $P' \| P$ and show that this correspondence is a simulation relation. A state s' in P' corresponds to a state (s_1, s_2) in $P' \| P$ iff the following conditions hold:

1. The cache states match, that is

(a) if s' is in *invalid* state, then s_1 and s_2 are in *invalid* state.

(b) if s' is in *shared-unmodified* state, then at least one state s_1 or s_2 is in *shared-unmodified* state and the other one is in *invalid* or *shared-unmodified*.

```
ASSIGN
init(cmd) := idle;
next(cmd) :=
 case
  state = I & !master :
   {copyback, read_shared, read_modified, idle};
  state = EM & !master :
   {copy_back, read_modified, read_shared, idle};
  state = EU & !master :
   {copy_back, read_modified, read_shared, idle};
  state = SU & !master :
   {invalidate, copy_back, idle};
  master : cmd;
  1 : idle;
 esac;
```

Figure 15.5
Command part for the invariant P'.

(c) if s' is in *exclusive-modified* state, then exactly one of the components is in *exclusive-modified* state and the other one is in *invalid*.

(d) if s' is in *exclusive-unmodified* state, then exactly one of the components is in *exclusive-unmodified* state and the other one is in *invalid*.

2. s has the master bit set to 1 if and only if exactly one of the states s_1 or s_2 has its master bit set to 1.

3. The value of the command variable cmd in the state s is the same as the value of the variable cmd in the state that has its master bit set to 1. Thus, if `master` $= 1$ in s_1, then the value cmd in s_1 should match the value of cmd in s. Similarly, if `master` $= 1$ in s_2, then the value cmd in s_2 should match the value of cmd in s.

It is straightforward to check that the initial states correspond and that for every pair of corresponding states s and (s_1, s_2), every possible transition from (s_1, s_2) is also possible from s.

We will check this fact for a specific pair of states. For other cases the analysis is similar. Consider the case where s is in the *exclusive-modified* state, s_1 is in the *exclusive-modified* state, and s_2 is in the *invalid* state. Thus, $s_1 \in P'$ can issue either `copy-back` or `read-modified` or `read-shared` commands, whereas $s_2 \in P$ can issue either a `read-shared` or a `read-modified` command. We will consider some of the transitions from the state (s_1, s_2) and show that there are corresponding transitions from the state s.

- Let `master` $= 1$ and `cmd = read-shared` in the state s_2. Recall that by issuing a `read-shared` command the processor gets a readable copy of the cache line. This happens when the second processor issues a `read-shared` command. Let (s_1', s_2') be the next state in $P' \| P$. In s_2' the state of the cache is *shared-unmodified* and in s_1' the state is *shared-unmodified* or *invalid*. Because the states s and (s_1, s_2) correspond, the `cmd` in the state s is also `read-shared`. Let s' be the successor state of s in P'. Thus, in s' the state of the cache is *shared-unmodified*. Hence, the states s' and (s_1', s_2') correspond.
- Let `master` $= 1$ and `cmd = read-modified` in s_2. Recall that by issuing a `read-modified` command the processor gets an exclusive copy of the cache line. Let (s_1', s_2') be the successor state of (s_1, s_2) in $P' \| P$. The state of the cache in s_1' is *invalid* and the state in s_2' is *exclusive-modified* or *exclusive-unmodified*. In the invariant process P' it is possible to issue a `read-modified` command and move to the *exclusive-modified* or *exclusive-unmodified* state. Therefore, the next state of s corresponds to (s_1', s_2').
- The cases when s_1 has `master` $= 1$ and issues either a `copy-back` command, a `read-shared` command, or a `read-modified` command are similar to the preceding cases.

15.4 Graph and Network Grammars

An important question in the study of families of Kripke structures is: How does one generate the infinite family? Most authors consider standard topologies such as rings or stars. We present a formalism based on graph grammars that lets us generate many interesting topologies.

Our treatment is based on the material in [246]. A graph over Σ (the *node alphabet*) and Δ (the *edge alphabet*) is a triple (N, ϕ, ψ), where N is a finite nonempty *set of nodes*, $\phi : N \to \Sigma$ is the *node labeling function*, and $\psi \subseteq N \times \Delta \times N$ is the *edge labeling function*. Let $\mathcal{G} = \{ D \mid D$ is a graph over Σ and $\Delta \}$; a *graph language* $\mathcal{D}$ over Σ and Δ is a subset of $\mathcal{G}$. A *context-free graph grammar* (CFGG) is a 5-tuple $G = (\Sigma_n, \Sigma_t, \Delta, \mathcal{S}, \mathcal{R})$ where the *nonterminal node alphabet* (Σ_n), the *terminal node alphabet* (Σ_t), and the *edge alphabet* (Δ) are finite nonempty mutually disjoint sets, $\mathcal{S} \in \Sigma_n$ is the *start label*, and $\mathcal{R}$ is a finite nonempty set of *production rules*. Each element in $\mathcal{R}$ is a quadruple $r = (A, D, I, O)$, where

1. $A \in \Sigma_n$;
2. $D = (N, \phi, \psi)$ is a connected graph over $\Sigma = \Sigma_n \cup \Sigma_t$ and Δ. The set Σ is the *entire node alphabet*;
3. $I \in N$ is the *input node*;
4. $O \in N$ is the *output node*.

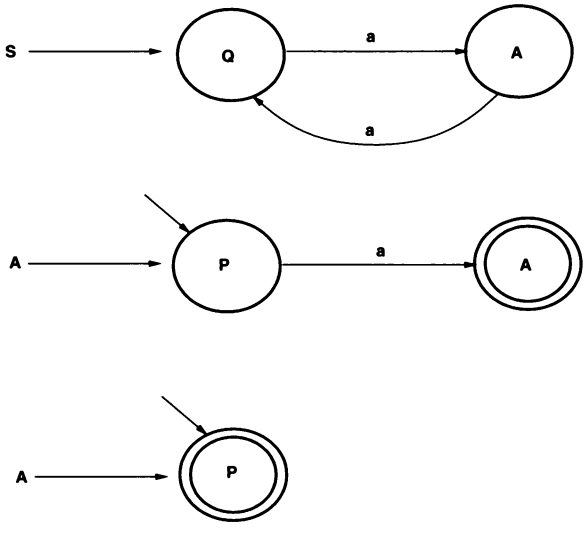


Figure 15.6
Rules for the graph grammar.

Given a graph whose nodes are labeled, a new graph is derived using one of the rules of the grammar. We start with a graph with a single node whose label is the start symbol $\mathcal{S}$. During a derivation, a node with label A is replaced by the graph D in some derivation rule (A, D, I, O). Every arc originally entering (exiting) the node labeled by A becomes an arc entering (exiting) the input node I (output node O).

Example Consider the grammar $G = (\Sigma_n, \Sigma_t, \Delta, \mathcal{S}, R)$ with $\Sigma_n = \{\mathcal{S}, A\}$, $\Sigma_t = \{P, Q\}$, and $\Delta = \{\mathrm{a}\}$. The rules are shown in Figure 15.6. The grammar generates all rings of the form $Q\ P^i$. The input nodes are indicated by an arrow, and the output nodes are indicated by double circles. Derivation of ring of size 3 is shown in Figure 15.7. Consider the second step. Because the node labeled with P is the input node, the arc from Q enters P. Similarly, the node labeled A has an arc going out to Q because it is the output node.

A *network grammar* is like a graph grammar except that the nodes in the graphs derived using the network grammar correspond to Kripke structures. The semantics of a derived graph is the Kripke structure obtained by composing the structures in all of its nodes. For example, if in the example given in Figure 15.6 we interpret P, Q as the processes in Figure 14.1 and the edges as composition operators, we can generate the infinite family $\mathcal{F} = \{Q \| P^i\}_{i=1}^{\infty}$ of token rings. Network grammars have been used to perform induction on the topological structure of the network [68, 187, 229].

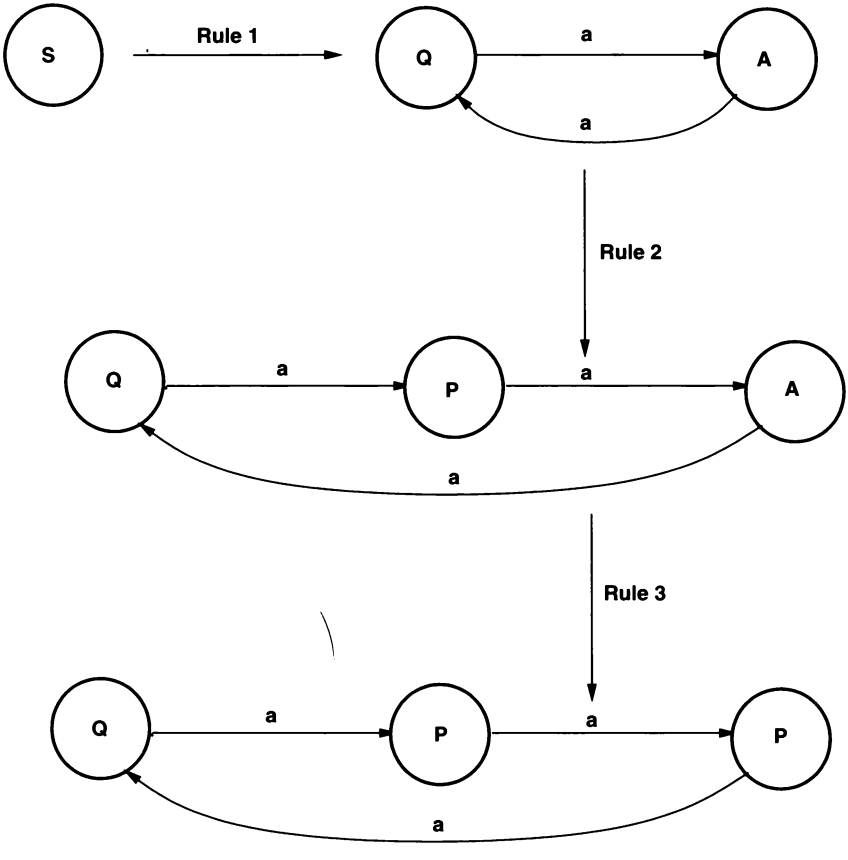


Figure 15.7
Derivation of a ring of size 3.

In [68, 187, 229], the family $\mathcal{F}$ is defined by means of a network grammar. The rules of the grammar define inductively the legal configurations in the family, where a configuration is given as a communication graph with an assignment of basic processes (Kripke structures) to nodes of the graph. Based on the rules of the network grammar, induction on the topological structure of the network is performed to establish an invariant for the entire family. We will explain these techniques by an example. Consider the network grammar G in Figure 15.8 that generates an infinite family of binary trees of depth ≥ 2. The symbols `root`, `inter`, `leaf` are the terminal processes. A system that checks parity based on this grammar is discussed later in the chapter.

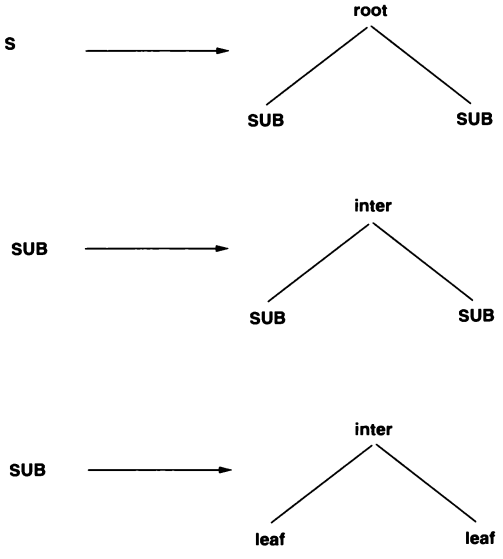


Figure 15.8
The network grammar G for binary trees.

For simplicity, we will use a linear representation of the network grammars in the remainder of this chapter. For example, the second rule for SUB in Figure 15.8 will be written as

$$\text{SUB} \longrightarrow \texttt{inter} \| \texttt{leaf} \| \texttt{leaf}$$

In order to verify a family of Kripke structures derived by a network grammar we extend the invariant rule presented in Section 15.2. With each of the nonterminals in the network grammar we associate an invariant that will be greater than any of the Kripke structures derived from this nonterminal. As before, we will assume that $\|$ is monotonic with respect to $\geq$.

To illustrate our idea, let $inv(\text{SUB})$ be the invariant associated with the non-terminal SUB in the network grammar for binary trees. This invariant must satisfy the following *monotonicity* conditions:

$$inv(\text{SUB}) \geq \texttt{inter} \| inv(\text{SUB}) \| inv(\text{SUB}) \tag{15.1}$$

$$inv(\text{SUB}) \geq \texttt{inter} \| \texttt{leaf} \| \texttt{leaf} \tag{15.2}$$

Notice that the two equations correspond to the last two rules in the grammar. Now we prove that $inv(\text{SUB})$ is larger than any process derived by the nonterminal SUB in the

ordering $\geq$. Our proof uses induction on the number of steps in a derivation. We use the symbol SUB $\overset{k}{\Rightarrow}$ w to denote that w is derived from SUB using k steps. The result is true for $k = 1$ because of Equation 15.2 given earlier. Let w be derived from SUB using $k > 1$ derivations. The process w has the following form:

$$w = \texttt{inter} \| w_1 \| w_2$$

The processes w_1 and w_2 are derived using less than k derivations, so by the induction hypothesis we have the following equations:

$$inv(\text{SUB}) \geq w_1$$

$$inv(\text{SUB}) \geq w_2$$

$$\texttt{inter} \| inv(\text{SUB}) \| inv(\text{SUB}) \geq \texttt{inter} \| w_1 \| w_2$$

The third equation follows from the first two using monotonicity of the composition operator with respect to $\geq$. Using Equation 15.1 and the equation given above, we get that $inv(\text{SUB}) \geq w$. Therefore, the process $\mathcal{I}$ given below is an invariant (using the partial order $\geq$) for the infinite family generated by the grammar.

$$\mathcal{I} = \texttt{root} \| inv(\text{SUB}) \| inv(\text{SUB})$$

In Shtadler and Grumberg [229], a specific process generated by the nonterminal SUB is used as an invariant. This invariant is required to be equivalent to all other Kripke structures that can be derived from SUB. An abstraction based on the specification is used in [68] to construct an invariant. Next, we describe the method to derive invariants presented in [68].

We consider a family of binary trees in which each leaf has a bit value. We verify an algorithm that computes the parity of the values at the leaves. The algorithm is taken from [242] and works as follows. The root process initiates a wave by sending the *readydown* signal to its children. Every internal node that gets the signal sends it to its children. When the signal *readydown* reaches a leaf process, the leaf sends the *readyup* signal and its *value* to its parent. An internal node that receives the *readyup* and *value* from both its children, sends the *readyup* signal and the $\oplus$ of the values received from the children to its parent. When the *readyup* signal reaches the root, one wave of the computation is terminated and the root can initiate another wave. The structure of the network derived from the grammar G is given schematically in Figure 15.9. For example, the inputs *readyup_l* and *value_l* of an internal node are identified with the outputs *readyup* and *value* of its left child.

Next, we describe the various processes and their signals in detail. First, we describe the process `inter`. The process `inter` is the process corresponding to an internal node of the tree. The various signals for the process are shown in the table in Figure 15.10. The state

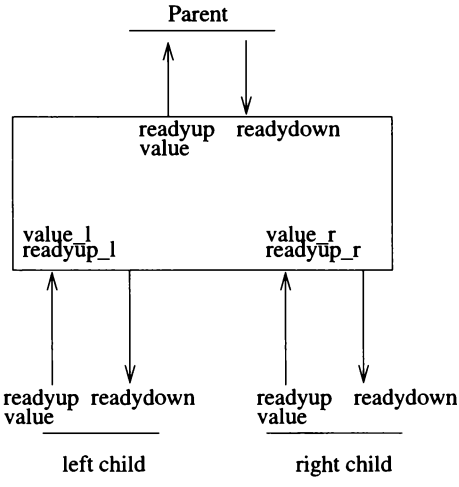


Figure 15.9
Internal node of the tree.

State variables	Output variables	Input variables
root_or_leaf	*readydown*	*readydown*
readydown	*readyup*	*readyup_l*
readyup_l	*value*	*readyup_r*
readyup_r		*value_l*
value		*value_r*
readyup		

Figure 15.10
The signals for process `inter`.

variables are internal variables that are used to preserve the value of the input variables. The input and the output variables provide the interface with the environment.

The following equations are invariants for the state variables:

$$root_or_leaf = 0$$

$$readyup = readyup_l \wedge readyup_r$$

Note that $root_or_leaf = 0$ because this is an internal node. The output variables have the same value in each state as the corresponding state variable, for example, the output variable *readydown* has the same value as the state variable *readydown*. The equations

given below show how the input variables affect the state variables. In the equations given below, the primed variables on the left hand side refer to the next state variables and the right hand side refers to the input variables.

$$readydown' = readydown$$

$$readyup_l' = readyup_l$$

$$readyup_r' = readyup_r$$

$$value' = (readyup_l \wedge value_l) \oplus (readyup_r \wedge value_r)$$

Because the `root` process does not have a parent, it does not have the input variable *readydown*. The invariant $root_or_leaf = 1$ is maintained for the root and the leaf process. Because the leaf process does not have a child, the output variable *readydown* is absent. The leaf variable has only one input variable *readydown* and the following equation between the next state variables and input variables is maintained:

$$readyup' = readydown$$

This equation holds for leaf nodes because they send a *readyup* signal immediately after they get the *readydown* signal. For each `leaf` process the assignment for the state variable *value* is decided nondeterministically in the initial state and then kept the same throughout the computation.

A state in the basic processes `(root,leaf,inter)` is a specific assignment to the state variables. We call the set of such states Σ. Notice that the state set is $\Sigma \cong \{0, 1\}^6$ because there are six state variables. Let $value_1, \cdots, value_n$ be the values in the n leaves. Let *value* be the value calculated at the root. Because at the end of the computation the root process should have the parity of the bits $value_i$ $(1 \leq i \leq n)$, the following equation should hold at the end of the computation:

$$value \oplus \bigoplus_{i=1}^{n} value_i = 0. \tag{15.3}$$

Let p be a new proposition that is true of all states in Σ that satisfy $root_or_leaf \wedge value$. The proposition p will be true at any root or leaf node that has bit value 1. The proposition $not(p)$ is the complement of p and is true in states of internal nodes and in states of root or leaf nodes with value bit 0. Notice that the state set of `inter`||`leaf`||`leaf` is Σ^3. In general, a tree consisting of n processes (the processes are from the set {`root`, `inter`, `leaf`}) has the state set Σ^n. Therefore, the state set of the entire family of parity trees is $\bigcup_{i=1}^{\infty} \Sigma^i$, which is a subset of Σ^*.

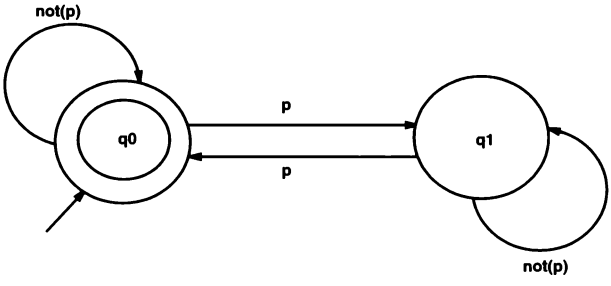


Figure 15.11
Automaton for parity.

In order to reason about the entire family of parity trees, we need to have a formalism which accepts states from the set Σ^*. In section 15.1 we show how this can be done using regular expressions. For efficiency concerns we use instead deterministic finite-state automata over the alphabet Σ. The finite-state automata will perform the role of atomic propositions in the logic ACTL.

The automaton given in Figure 15.11 accepts the strings in Σ^* that satisfy Equation 15.3. Since $root_or_leaf = 0$ for internal nodes, the automaton essentially ignores the values at the internal nodes.

Unlike the notation for graph grammars, here an arrow indicates an initial state and a double circle indicates an accepting state. This notation is standard for finite automata. We also want to assert that every process is finished with its computation. This is signaled by the fact that $readyup = 1$ for each process. The automaton given in Figure 15.12 accepts a string $w \in \Sigma^*$ iff $readyup$ is true in each letter of w (notice that each letter in $w \in \Sigma^*$ corresponds to a state in a component), that is, all processes have finished their computation. We use the product of these two automata as our atomic formula. We use $\mathcal{P}$ to denote the product automaton. Let Q be the set of states of the product automaton, $\delta : Q \times \Sigma \rightarrow Q$ the next state function, and $s_0 = (m_0, q_0)$ the initial state. The state (m_0, q_1) of the product automaton has the semantics that the computation is finished, but the parity is incorrect. We call the state (m_0, q_1) bad. We want to check that every reachable state $\sigma \in \Sigma^*$ of the family of parity trees satisfies the condition that if the computation is finished in that state, then the *root* process has the correct parity, that is, $\delta(s_0, \sigma) \neq bad$.

Each automaton with the alphabet Σ introduces an abstraction on the set of states Σ^* of the family of parity trees. We will first describe the abstraction function h on the state set Σ of the basic processes `root`, `leaf`, and `inter`. Consider a state $a \in \Sigma$. The abstraction $h(a)$ of a is the function that a induces on the state set Q of the product automaton. Thus, $h(a) : Q \rightarrow Q$ where $h(a)(q) = \delta(q, a)$ and δ is the transition function of the automaton.

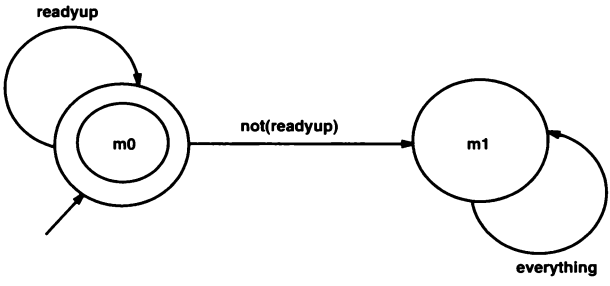


Figure 15.12
Automaton for ready.

Now consider an arbitrary state $\sigma = (a_0, a_1, \cdots, a_k) \in \Sigma^k$. The abstraction of σ, $h(\sigma) : Q \rightarrow Q$ is given by the following equation:

$$h(\sigma) = h(a_0) \circ h(a_1) \circ \cdots \circ h(a_k)$$

where symbol $\circ$ denotes the function composition.

We will say that σ_1 is equivalent to σ_2 if and only if their abstractions are equal (as functions), that is, $h(\sigma_1) = h(\sigma_2)$. Note that, the number of different functions that can be induced by some state $\sigma \in \Sigma^*$ is bounded by $|Q|^{|Q|}$. Thus we mapped the infinite state space Σ^* to a finite abstract domain. In practice, the number of different abstract values to which reachable states are mapped will be much smaller.

Example Consider a state $a_0 \in \Sigma$ in which p is true and *readyup* is true. Then $h(a_0)$ is the following function:

$$h(a_0)(q0, m0) = (q1, m0)$$
$$h(a_0)(q0, m1) = (q1, m1)$$
$$h(a_0)(q1, m0) = (q0, m0)$$
$$h(a_0)(q1, m1) = (q0, m1)$$

To see why this is true, consider, for example, $h(a_0)(q_0, m_0)$. In the automaton given in Figure 15.11, there is a transition on a_0 from the state $q0$ to $q1$. Likewise, in the second automaton there is a transition from the state $m0$ to $m0$ on a_0. Therefore, $h(a_0)(q0, m0)$ is $(q1, m0)$.

Consider another state $a_1 \in \Sigma$ in which `not(p)` is true and *readyup* true. The abstraction $h(a_1)$ is the following function:

$h(a_1)(q0, m0) = (q0, m0)$

$h(a_1)(q0, m1) = (q0, m1)$

$h(a_1)(q1, m0) = (q1, m0)$

$h(a_1)(q1, m1) = (q1, m1)$

The abstraction of the state (a_0, a_1) is $h(a_0) \circ h(a_1)$.

The abstract process corresponding to P is denoted by $h(P)$. There is a transition from the abstract state h_1 to the abstract state h_2 in $h(P)$ if and only if there exist two states s_1 and s_2 in P such that $h(s_1) = h_1$, $h(s_2) = h_2$, and there exists a transition from s_1 to s_2 in P. Given two processes P_1 and P_2, we say that $P_1 \preceq P_2$ if and only if there exists a relation $\mathcal{E}$ between the states of P_1 and P_2 such that the following conditions hold for all $(s, s') \in \mathcal{E}$:

- $h(s) = h(s')$.
- Given a state s_1 in P_1 and a transition $s \xrightarrow{a} s_1$ in P_1, there exists a transition $s' \xrightarrow{a} s_1'$ in P_2 such that $(s_1, s_1') \in \mathcal{E}$.

Abstraction can also be applied to abstract states $h(h_1) = h_1$ when h_1 is an abstract state. This definition differs from the one given in Chapter 11 in two respects:

- Related states have to agree on their abstraction rather than atomic propositions.
- The transitions are labeled by action symbols and the corresponding transitions have to agree on their labeling.

Given a process P and the corresponding abstract process $h(P)$ define a relation $\mathcal{E}_h$ between the state sets of P and $h(P)$ in the following manner:

$(s, h_1) \in \mathcal{E}_h \Leftrightarrow h(s) = h_1$

Using the relation $\mathcal{E}_h$ one can prove that $h(P) \succeq P$. The abstract composition of two processes P_1 and P_2 is defined as follows:

$P_1 \|_h P_2 = h(P_1 \| P_2)$

Let h be the abstraction function induced by the product automaton. Let $\|_h$ be the abstract composition operator and $\preceq$ the *simulation* relation. Let I_1, I_2 be abstract processes defined as follows:

$I_1 = h(inter) \|_h h(leaf) \|_h h(leaf)$

$I_2 = h(inter) \|_h I_1 \|_h I_1$

The following equations can be checked automatically:

$h(inter)\|_h I_1\|_h I_1 \not\preceq I_1$

$I_1 \preceq I_2$

$h(inter)\|_h I_2\|_h I_2 \preceq I_2$

From the first equation given above it is clear the I_1 cannot be used as an invariant for the nonterminal SUB. If we select $inv(\text{SUB}) = I_1$, the induction step corresponding to the second rule of the grammar does not hold.

SUB $\rightarrow$ `inter`$\|$SUB$\|$SUB

Notice that I_2 was derived from the second rule of the grammar by substituting I_1 for SUB in the right hand side of the rule. Suppose we use $inv(\text{SUB}) = I_2$ and $inv(\mathcal{S}) = h(\text{root})\|_h I_2\|_h I_2$ as the invariants for the nonterminals. From the equations given above the following inequalities can be derived:

$inv(\text{SUB}) \succeq h(\text{inter})\|_h inv(\text{SUB})\|_h inv(\text{SUB})$

$inv(\text{SUB}) \succeq h(\text{inter})\|_h h(\text{leaf})\|_h h(\text{leaf})$

After checking the monotonicity conditions, we can conclude that $H = h(root)\|_h I_2\|_h I_2$ simulates all the networks generated by the context-free grammar G. After we have constructed H, we can check that all reachable states in H have the desired property. By Theorem 16 given in Chapter 11, we have the result that every network derived using G has the desired property, that is, when the computation is finished the root process has the correct parity. We also checked that from each state we must always reach a state where the computation is finished and is correct, that is, $\mathbf{AF}\mathcal{P}$.

15.5 Undecidability Result for a Family of Token Rings

In this section we prove the undecidability of the verification problem for infinite families of finite-state systems mentioned at the beginning of the chapter. The reader can safely skip this section when reading the chapter for the first time.

Following Suzuki [237] we show how to simulate a Turing machine T by a family of bidirectional rings. A ring of size n simulates n steps of the Turing machine on an empty tape. If the Turing machine halts within n steps, then some process in the ring will enter a special *halt* state and remain there forever. If the Turing machine does not halt within n steps, then no process will ever enter the *halt* state. Thus, the Turing machine does not halt

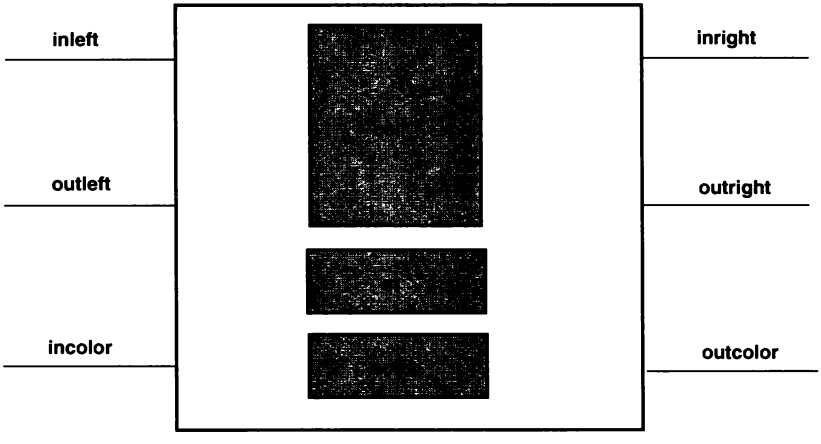


Figure 15.13
Process P_i.

on the empty tape iff every ring in the family satisfies the formula $\mathbf{AG} \bigwedge_i \neg halt_i$, where $halt_i$ is true if process i is in the *halt* state.

The Turing machine T is a 5-tuple $T = (Q, \Sigma, \delta, q_0, halt)$, where Q is the set of states, Σ is the tape alphabet, $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{left, right\}$ is the transition function, q_0 is the initial state, and *halt* is the final state. A ring that simulates n steps of T consists of n processes $P_0, \ldots, P_{n-1}$, each of which represents one cell of the Turing machine tape. We assume that the Turing machine T has a one-way infinite tape extending to the right. Hence, within n steps, it can scan at most n cells of its tape.

Assume that T scans symbol a in cell i when it is in state q. Then process P_i will be in a particular state that represents the combination of symbol a and state q. Process P_i will simulate one move of T and will send the new state q' to the appropriate neighbor according to the move of T. A diagram for process P_i is shown in Figure 15.13. Process P_i is connected to process P_{i-1} on its left and to process P_{i+1} on its right, where $i + 1$ and $i - 1$ are computed modulo n. Input $inright_i$ is connected to $outleft_{i+1}$. Outputs $outright_i$ and $outcolor_i$ are connected to $inleft_{i+1}$ and $incolor_{i+1}$, respectively.

We assume a synchronous model of computation in which every process makes a step at each time and in which the values of the outputs at a certain step are the values of the corresponding inputs in the next step.

The current state of P_i is determined by the value of its variables *cell*, *st* and *color*, which range over Σ, Q, and $\{white, black\}$, respectively. Initially, all variables *cell* are blank; $st_0 = q_0$ and for all $i > 0$, $st_i = null$; $color_0 = black$ and for all $i > 0$, $color_i =$

```
while true do
    if incolor ≠ null and st = halt then
        while true do outright := outleft := null;
    end if;
    if incolor ≠ null and st ≠ null
            and δ(st, cell) = (q', a', d) then
        cell := a';
        outright := if d = right then q' else null;
        outleft := if d = left then q' else null;
    else
        outright := outleft := null;
    end if;
    st := if inright ≠ null then inright else inleft;
end while;
```

Figure 15.14
Simulation program for process P_i, $i > 0$.

```
while incolor ≠ black do
    if incolor = null then
        outcolor := null;
    if incolor = white then
        outcolor := black;
end while;
while true do outcolor := null;
```

Figure 15.15
Counting program for process P_0.

white. Also, $outcolor_0 = white$ while all other outputs (and the corresponding inputs) are initially *null*.

The computation on the ring consists of two phases that run in lockstep. One phase (shown in Figure 15.14) simulates steps of the Turing machine T on the empty tape while the other phase (shown in Figure 15.15 and 15.16) counts until n and then stops the simulation.

The counting phase transfers a colored token around the ring. In each of the n rounds of this phase, the token is propagated from P_0 back to P_0. Initially, all processes are *white*, except P_0, which is *black*. In addition, all processes have $outcolor = null$ except P_0, which

```
while true do
    if incolor ≠ null then
        outcolor := color;
    else
        outcolor := null;
    end if;
    if (incolor = black and color = white) then
        color := black;
end while;
```

Figure 15.16
Counting program for process P_i, $i > 0$.

has $outcolor = white$. When a process gets a *null* token, it passes the token unchanged to its right neighbor. Similarly, if it gets a token that has the same color as it does, it sends the token unchanged to the right. If a process gets a *black* token when its color is *white*, it changes it own color to *black* and sends a *white* token to the right.

Process P_0 behaves somewhat differently. Its color is always *black*. In the first round it sends *white* to its right neighbor. If it receives a *null* token, then it sends a *null* token to the right. When it receives a *white* token, it sends a *black* token to the right. Finally, when it receives a *black* token, it changes to an idle phase in which it sends *null* forever. Thus, in any round one more process turns *black* by getting a *black* message from its *black* neighbor on the left. When P_0 gets a *black* message from P_{n-1}, exactly n steps of T have been simulated and the ring moves to an idle phase.

The simulation phase makes sure that exactly one step is performed in each round of the counting phase by activating the appropriate process only when it gets a token (either *black* or *white*). When the head of the Turing machine scans cell i in control state q, P_i has $st = q$ while all other processes have $st = null$. When P_i gets the token it simulates $\delta(st, cell) = (q', a', d)$ by setting $cell := a'$, $st := null$ and by sending q' to either its left neighbor or its right neighbor, according to the direction d. The first step of T, $\delta(q_0, blank) = (q', a', right)$, is simulated by P_0 regardless of the value of $incolor_0$.

A subtle case occurs when some P_i has $st \neq null$ and $incolor \neq null$ and the Turing machine moves to the right. P_i then simulates one step of the Turing machine by setting $outright = q'$ in order to propagate the new state to P_{i+1}. It also propagates the token to P_{i+1} by setting the value of *outcolor* appropriately. The program ensures that P_{i+1} will not simulate another step of the Turing machine until the next round by having P_{i+1} first check

the value of *incolor* together with the *old* value of its variable *st*. Later, P_{i+1} updates the value of *st* according to $inleft_{i+1}$ (which is identical to $outright_i$).

Next, we will describe invariants to establish the correctness of our simulation. There are n rounds in the computation. We number rounds from 0 to $n-1$. Each round has n steps, and we number the steps from 0 to $n-1$ as well. Round i simulates the i-th move of the Turing machine. The following properties can be proved about the computation.

- After step $i-1$ of round i, process P_i changes color from white to black.
- At the end of round i, state of process P_j is equal to $q \neq null$ iff after i moves the Turing Machine is in state q and is scanning the j-th cell.
- Assume that the process P_j is in state q after round i. All other processes are in state *null*. At step $j-1$ of round $i+1$ the process P_j receives a nonnull color from its neighbor. P_j sends the appropriate state to its left or right neighbor (depending on whether the Turing Machine moves left or right) and then sets its state to *null*. Notice that P_j can only simulate a move if it receives a non-null color.

Although the problem is undecidable in general, for *specific* families it may be solvable.

16 Discrete Real-Time and Quantitative Temporal Analysis

Computers are frequently used in critical applications where predictable response times are essential for correctness. Such systems are called *real-time systems*. Examples of such applications include controllers for aircraft, industrial machinery and robots. Due to the nature of such applications, errors in real-time systems can be extremely dangerous, even fatal. Guaranteeing the correctness of a complex real-time system is an important and nontrivial task. Because of this, only conservative and usually ad hoc approaches to design and implementation are routinely used.

Other factors make the validation of real-time systems particularly difficult. The architecture of computer applications is becoming extremely complicated. As a system increases in complexity, so does the probability of introducing an error. Moreover, performance is becoming an important factor in the success of new applications. Due to competition, new products have to fully utilize the available resources. A slow component can affect the performance of the whole system. Consequently, the task of verifying that new applications satisfy their timing specifications is more critical than ever before.

16.1 Real-Time Systems and Rate-Monotonic Scheduling

Because real-time systems are used in critical application, conservative approaches have been traditionally used in their design. This has frequently led to simple but inefficient implementations. An example of such a technique is *static time-slicing*, which divides time equally among all tasks. Each task executes until its time slot has been used and then releases the processor. The resulting program is easy to analyze, but rather inefficient, since all tasks are given equal resources, regardless of their importance or resource utilization. Recently, more powerful techniques to analyze the behavior of a real-time system have been developed. *Rate-monotonic scheduling* theory (RMS) [172, 176, 228] is an example. The RMS theory is applicable to systems described by a set of periodic tasks. Each task corresponds to a concurrent process of the system and is characterized by its periodicity (how often it executes) and its execution time at each instantiation. RMS consists of two components. The first is an algorithm for assigning higher priorities to processes with shorter periods. Optimal response time with respect to static priority algorithms is guaranteed by the RMS theory if priorities are assigned according to this rule [176]. The second component of the RMS theory is a schedulability test based on total CPU utilization; a set of processes (which have priorities assigned according to RMS) is schedulable if the total utilization is below a computed threshold. If the utilization is above this threshold, schedulability is not guaranteed.

RMS is a powerful tool for analyzing real-time systems. It is simple to use, yet it provides very important information for designers. However, this analysis imposes a series

of restrictions on the system being verified. Only certain types of processes are considered, with limitations, for example, on periodicity and synchronization. Recent work has extended this theory to more general classes of processes, but limitations still exist [131]. RMS can handle only systems that can be described within the theory. Moreover, the kinds of properties that can be verified are also restricted to properties that can be modeled as task execution times. Verifying distributed systems or systems that do not have a regular communication pattern is not a trivial task in general. In addition, checking for properties that cannot be easily expressed as task execution times such as the number of occurrences of arbitrary events in the system can also be complex.

16.2 Model Checking Real-Time Systems

It is possible to use symbolic model checking to verify discrete real-time systems. However, the model checking tools described previously in this book are not suitable to perform this type of verification. It is difficult, for example, to express complex timing properties. It is possible to express the property that "event p will happen in the future," but it is not simple to express the property that "event p will happen within at most n time units" without using the next time operator in convoluted ways. Moreover, quantitative information such as response time or the number of occurrences of events cannot be directly obtained using these techniques. Temporal-logic model checking cannot be used in a natural and efficient way to verify many types of real-time systems that occur frequently in practice.

16.2.1 Related Methods for Verifying Real-Time Systems

Other approaches to schedulability analysis include algorithms for computing the set of reachable states of a finite-state system [57, 117, 122]. A model for the real-time system is constructed with the added constraint that whenever an exception occurs (e.g., a deadline is missed) the system transitions to a special exception state. Verification consists of computing the set of reachable states and checking whether the exception state is in this set. Unlike RMS, no restrictions are imposed on the model in this approach, but the algorithm only checks if exceptions can occur or not. Other types of properties cannot be verified, unless encoded in the model as exceptions. Even though most properties can be encoded as exceptions, this can sometimes be difficult and error-prone. Symbolic model-checking techniques have also been extended to handle real-time systems [76, 78, 251]. However, these methods as well as the others mentioned only determine if the system satisfies a given property, and do not provide detailed information on its behavior. Restricted quantitative analysis on discrete-time models can be performed [79], but only to the extent of computing minimum/maximum delays.

In this chapter we describe a method for specifying and verifying *discrete* real-time systems, which is compatible with symbolic model checking techniques and can handle large systems [49]. Furthermore, algorithms derived from symbolic model checking are used to compute quantitative information about the model. An important benefit of this approach is that the information produced allows the user to check whether the model satisfies various real-time constraints: schedulability of the tasks of the system can be determined by computing their response time; reaction times to events and several other parameters of the system can also be analyzed by this method. This information provides insight into the behavior of the system and in many cases it can help to identify inefficiencies and suggest optimizations to the design. The same algorithms can then be used to analyze the performance of the modified design. The evaluation of how the optimizations affect the design can be done before the actual implementation. This can significantly reduce development costs.

An important characteristic of this method is that it counts the number of computation steps between events or the number of occurrences of events in an interval. Because of this, it finds application in synchronous systems such as computer circuits and protocols. Real-time systems usually do not execute in lock-step and might not seem to be appropriate for our method. However, they are frequently subject to tight timing constraints, which are difficult to satisfy using asynchronous design techniques. Furthermore, programmers often try to reduce asynchronous behavior in their designs in order to ensure predictability. As a result, real-time systems can often be analyzed using techniques based on discrete time. Some systems, however, are inherently asynchronous in nature. For these systems more complex verification techniques based on continuous time are necessary. We will discuss methods for verifying continuous real-time systems in the next chapter.

16.3 RTCTL Model Checking

A simple and effective way to allow the verification of time bounded properties is to introduce bounds in the CTL temporal operators. The extended logic is called RTCTL [108]. The expressive power of RTCTL is the same as CTL, since the bounded operators can be translated into nested applications of the EX (or AX) operators. However, this translation is often impractical, and RTCTL provides a much more compact and convenient way of expressing such properties.

The basic RTCTL temporal operator is the *bounded until* operator which has the form: $\mathbf{U}_{[a,b]}$, where $[a, b]$ defines the time interval in which the property must be true. We say that $f\ \mathbf{U}_{[a,b]}\ g$ is true of some path $\pi = s_0, s_1, \ldots$ if g holds in some future state s on the path, f

is true in all states between s_0 and s, and the distance from s_0 to s is within the interval $[a, b]$. The bounded **EG** operator can be defined similarly. Other temporal operators are defined in terms of these two operators. More formally, we extend CTL to include bounded versions of the operators **EU** and **EG** by adding the following clauses to the formal semantics of CTL:

1. $s \models \mathbf{E}[f\ \mathbf{U}_{[a,b]}\ g]$ if and only if there exists a path $\pi = s_0 s_1 s_2 \ldots$ starting at $s = s_0$ and some i such that $a \leq i \leq b$ and $s_i \models g$ and for all $j < i, s_j \models f$.
2. $s \models \mathbf{EG}_{[a,b]}\ f$ if and only if there exists a path $\pi = s_0 s_1 s_2 \ldots$ starting at $s = s_0$ and for all i such that $a \leq i \leq b, s_i \models f$.

As an example of the use of the bounded until, consider the property "It is always true that p may be followed by q within 3 time units." This property can be expressed in RTCTL as $\mathbf{AG}(p \rightarrow \mathbf{EF}_{[0,3]}\ q)$, where the bounded **EF** operator is derived from the bounded until just as in the unbounded case. That is, $\mathbf{EF}_{[a,b]}\ f \equiv [true\ \mathbf{U}_{[a,b]}\ f]$.

In order to verify properties written with this operator, we use a modification of the fixpoint computation that is used in CTL model checkers. It is easy to see that the formula $\mathbf{E}[f\ \mathbf{U}_{[a,b]}\ g]$ can be computed in the following manner:

$$\begin{cases} \mathbf{E}[f\ \mathbf{U}_{[a,b]}\ g] = f \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}_{[a-1,b-1]}\ g] & \text{if } a > 0 \text{ and } b > 0, \\ \mathbf{E}[f\ \mathbf{U}_{[0,b]}\ g] = g \vee (f \wedge \mathbf{EX}\,\mathbf{E}[f\ \mathbf{U}_{[0,b-1]}\ g]) & \text{else, if } b > 0, \\ \mathbf{E}[f\ \mathbf{U}_{[0,0]}\ g] = g & \text{otherwise} \end{cases}$$

Other operators are computed similarly.

16.4 Quantitative Temporal Analysis: Minimum/Maximum Delay

Traditional formal verification algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. These techniques do not provide any information about how much a system deviates from its expected performance. However, such information can be extremely useful in fine-tuning the behavior of the system. In this section we describe algorithms to compute quantitative timing information, such as exact minimum and maximum delays (in terms of the number of transitions) between a request and the corresponding response. The algorithms are designed to work well with symbolic techniques based on the use of binary decision diagrams and are very efficient in practice.

```
procedure min(start, final)
    i := 0;
    Z := start;
    Z' := T(Z) ∪ Z;
    while ((Z' ≠ Z) ∧ (Z ∩ final) = ∅) do
        i := i + 1;
        Z := Z';
        Z' := T(Z') ∪ Z';
    end while;
    if (Z ∩ final ≠ ∅) then
        return i;
    else return ∞;
    end if;
end procedure
```

Figure 16.1
Minimum delay algorithm.

16.4.1 Minimum Delay Algorithm

The algorithm takes as input a Kripke structure $M = (S, R, L)$ and two sets of states $start$ and $final$. It returns the length of (i.e., number of edges in) a shortest path from a state in $start$ to a state in $final$. If no such path exists, the algorithm returns infinity. In the algorithm, the function $T(S)$ gives the set of states that are successors of some state in S. In other words, $T(S) = \{s' \mid R(s, s') \text{ holds for some } s \in S\}$. In addition, the variables Z and Z' represent sets of states in the algorithm.

The first algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from $start$. If at any point, we encounter a state satisfying $final$, we return the number of steps taken to reach the state.

16.4.2 Maximum Delay Algorithm

This algorithm also takes $start$ and $final$ as input. It returns the length of a longest path from a state in $start$ to a state in $final$. If there exists an infinite path beginning in a state in $start$ that never reaches a state in $final$, the algorithm returns infinity. The function $T^{-1}(S')$ gives the set of states that are predecessors of some state in S' (i.e., $T^{-1}(S') = \{s \mid R(s, s') \text{ holds for some } s' \in S'\}$). Z and Z' will once more be sets of states. Finally, we denote by not_final the set of all states that are not in $final$.

```
procedure max(start, final)
    i := 0;
    Z := True;
    Z' := not_final;
    while ((Z' ≠ Z) ∧ (Z' ∩ start ≠ ∅)) do
        i := i + 1;
        Z := Z';
        Z' := T^-1(Z') ∩ not_final;
    end while;
    if (Z := Z') then
        return ∞;
    else return i;
    end if;
end procedure
```

Figure 16.2
Maximum delay algorithm.

The upper bound algorithm is more subtle than the previous algorithm. In particular, it must return infinity if there exists a path beginning in $start$ that remains within not_final. A backward search from the states in not_final is more convenient for this purpose than a forward search. At the ith iteration the current *frontier* is the set of states that are the beginning of paths with i states completely in not_final. Initially, i is 0, and the frontier is not_final. We then compute the set of predecessors (in not_final) of the current frontier. Those states are the beginning of paths with $i + 1$ states completely in not_final.

The algorithm stops in one of two cases. Either Z' does not contain states from $start$ at stage i. Because it contained states from $start$ at state $i - 1$, the size of the longest interval in not_final from a state in $start$ is $i - 1$. Because the transition relation is total, this interval has a continuation to a state outside not_final, that is, to a state in $final$. Thus, there is a path of length i from $start$ to $final$ and the algorithm returns i. In the other case, a fixpoint is reached and Z still contains some state in $start$. Because the set Z is finite and each state in it has an outgoing edge to a state in Z, each state is the start of an infinite path within Z which is included in not_final. Thus, there is an infinite path in not_final from a state in $start$. In this case, the algorithm returns infinity.

Next, we argue that the algorithm terminates. Suppose that the condition $Z' \cap start \neq \emptyset$ is never violated. We will show that $Z' = Z$ eventually holds. It can be easily seen that if a state is in the ith frontier, it is also in the $i - 1^{\text{th}}$ frontier, since states that are the beginning of intervals with i states completely in not_final are also the beginning of intervals of

$i - 1$ states within not_final. Consequently, the frontier at each iteration is contained in the previous one. Because the initial frontier must be finite, there are only a finite number of proper inclusions between the state sets that characterize the frontiers. Therefore there must be a k such that the frontier at the kth iteration is the same as the frontier at the $k + 1^{\text{th}}$ iteration, and the loop cannot execute more than k times without $(Z = Z')$ becoming true.

In many situations we are interested not only in the length of a path leading from a set of starting states to a set of final states, but also in the number of states on the path that satisfy a given condition. Thus, we may wish to determine the minimum or maximum number of times a condition *cond* holds on any path from *start* to *final*. These algorithms are called *condition counting algorithms*. We give two examples of how they can be used to analyze the performance of systems. The first example is evaluating the performance of a bus in a complex hardware system. Consider the interval of time between asserting a bus request and the corresponding bus grant. It is important to be able to compute the number of times other transactions are issued in this interval, because this is a measure of the traffic on the bus. The second example is determining the amount of *priority inversion* in a real-time system. Priority inversion occurs when a higher priority process is blocked by the execution of a lower priority process [221]. In this case, *start* corresponds to the states where the higher priority process requests execution, *final* corresponds to the states where this process is granted execution, and *cond* characterizes the states where a lower priority process is executed, blocking the process with higher priority. Efficient BDD based implementations for these algorithms and additional examples are described in [49, 52].

16.5 Example: An Aircraft Controller

One of the most critical applications of real-time systems is in aircraft control. It is extremely important that time bounds are not violated in such systems. This section briefly describes an aircraft control system used in military airplanes. The example illustrates how timing constraints can be checked using the quantitative algorithms described in Section 16.4.

16.5.1 System Description

The control system for an airplane can be characterized by a set of sensors and actuators connected to a central processor. This processor executes the software to analyze sensor data and control the actuators. Our model describes this control program and defines its requirements to ensure that the airplane operating constraints are met. The requirements used are similar to those of existing military aircraft and are derived from those described in [177].

The aircraft controller is divided into systems and subsystems. Each system performs a specific task in controlling a component of the airplane. The most important systems are modeled, including

- Navigation: Computes aircraft position. Takes into account data such as speed, altitude, and positioning data received from satellites or ground stations.
- Radar Control: Receives and processes data from radars. It also identifies targets and target position.
- Radar Warning Receiver: This system identifies possible threats to the aircraft.
- Weapon Control: Aims and activates aircraft weapons.
- Display: Updates information on the pilot's screen.
- Tracking: Updates target position. Data from this system are used to aim the weapons.
- Data Bus: Provides communication between processor and external devices.

Each system is composed of one or more subsystems. Timing constraints for each subsystem are derived from factors such as required accuracy, human response characteristics, and hardware requirements. For example, the screen must be updated frequently enough so that motion appears continuous. To accomplish this, the update must occur at least once every 50 ms. The table in Figure 16.3 gives the subsystems being modeled, as well as their major timing requirements. The priority assignment will be explained subsequently.

Concurrent processes are used to implement each subsystem. Communication among the various processes is done indirectly. No data is shared directly by multiple subsystems. Processes communicate only through data servers called *monitor tasks*. Each system maintains a server process that accepts requests for data and returns the desired information. The various subsystems in each system update the data in the servers. Monitor tasks only accept requests, respond to them, and then enter a waiting state. They are assigned low priority, and priority inheritance is used to maintain predictability [50, 221].

With the exception of the weapon system, all other systems contain only periodic processes, which are scheduled to execute at the beginning of their period. When a process is granted the CPU, it acquires the data it needs through the monitor tasks, executes, updates information on its own data server and blocks waiting for its next execution period.

The weapon system contains a mixture of *periodic* and *aperiodic* processes. It is activated when the display keyset subsystem identifies that the pilot has pressed the firing button. This event causes the weapon protocol subsystem to be activated. It then signals the weapon aim subsystem that had been blocked. Weapon aim is then scheduled to be executed every 50 ms. It aims the aircraft weapons based on the current position of the target. It also decides when to fire and then starts the weapon release subsystem. The firing se-

System	Subsystem	Period	Exec.	% CPU	Priority
Display	status update	200	3	1.50	12
	keyset	200	1	0.50	16
	hook update	80	2	2.50	36
	graphic display	80	9	11.25	40
	store update	200	1	0.50	20
RWR	contact mgmt.	25	5	20.00	72
Radar	target update	50	5	10.00	60
	tracking filter	25	2	8.00	84
NAV	nav update	50	8	16.00	56
	steering cmds.	200	3	1.50	24
Tracking	target update	100	5	5.00	32
Weapon	weapon protocol	200[a]	1	0.50	28
	weapon aim	50	3	6.00	64
	weapon release	200[b]	3	1.50	98
Data bus	poll bus devices	40	1	2.50	68

a. Weapon protocol is an aperiodic process with a deadline of 200 ms.
b. Weapon release has a period of 200 ms, but its deadline is 5 ms.

Figure 16.3
Timing requirements for aircraft controller.

quence can be aborted until weapon release is scheduled, but not after this point. Weapon release then executes periodically and fires the weapons five times, once per second.

In order to enforce the different timing constraints of the processes, priority scheduling is used. Predictability is guaranteed by scheduling the processes using Rate Monotonic Scheduling (RMS) [171, 176].

16.5.2 Model of the Aircraft Control System

The aircraft control system has been modeled using the tool VERUS [51]. Model checking has been used to verify its functional correctness, while its timing correctness has been checked using the quantitative algorithms described previously. Most of the characteristics described above have been implemented, although some abstractions have been performed for simplicity. A more detailed description of the implementation follows.

The time for an atomic transition in the model is assumed to be one millisecond. A global timer controls the scheduling of periodic processes. Whenever awakened, a process requests execution and waits until it has been granted the CPU. For each process, an internal counter stores the elapsed execution time. After finishing execution, a process releases the

CPU and blocks, waiting for the next period. The time to request data from a monitor task and wait for the response is assumed to be small compared to the total execution time. This is reasonable if we assume an efficient implementation. Sending request and response messages takes only a small amount of time. Processing in the monitor tasks is also fast, considering the limited range of functions performed. The assumption can only be violated if multiple processes access the monitor simultaneously. The access pattern to the monitor tasks, however, minimizes this possibility. They simply receive requests, retrieve data from memory, and return it. There are no nested critical sections. Moreover, priority inheritance protocols [50, 221] have been used to maintain predictability and eliminate the possibility of unbounded blocking due to synchronization.

We consider two scheduling policies, preemptive scheduling and nonpreemptive scheduling. A preemptive scheduler accepts requests for execution and chooses the highest priority process requesting the CPU. If a request arrives from a higher priority process after execution has started, the scheduler preempts the executing process and starts the higher priority one. When a process finishes executing it resets its request, and the scheduler chooses another process. However, preemptability is a feature that may not always be available. With a nonpreemptive scheduler, once a process starts executing, it continues executing until it voluntarily releases the CPU. If a higher-priority process requests execution, it has to wait until the running process finishes. Nonpreemptive schedulers usually cause response time for higher-priority processes to be higher. They are, however, simpler to implement and allow for simpler programs. Modeling both types of schedulers allows us to compare the behavior of the system under different conditions. The results obtained can be used to assist in deciding whether preemption is necessary for system correctness in this case.

16.5.3 Verification Results

Schedulability is one of the most important properties of a real-time system. It states that no process will miss its deadline. In this example the deadlines are the same as the periods (except for the weapon release subsystem). The table in Figure 16.4 summarizes the execution times computed by the quantitative analysis performed. Processes are shown in decreasing order of priority. Deadlines are also shown so that schedulability can be easily checked. Minimum and maximum execution times are given for both preemptive and nonpreemptive schedulers.

We can see from the table in Figure 16.4 that the process set is schedulable using preemptive scheduling. From our results we can also identify many important parameters of the system. For example, the response time is usually very low for best-case computations, but it is also good for the worst case. Most processes take less than half of their permitted

		Execution Times			
		Preemptive		Non Preemptive	
Subsystem	Deadline	Min	Max	Min	Max
Weapon release	5	3	3	3	9
Radar tracking filter	25	2	5	2	10
RWR contact mgmt.	25	7	10	7	15
Data bus poll	40	1	11	1	14
Weapon aim	50	10	14	2	18
Radar target update	50	12	19	12	19
NAV update	50	20	34	20	27
Display graphic	80	10	44	10	43
Display hook update	80	14	46	14	47
Tracking target update	100	26	51	26	51
Weapon protocol	200	1	21	3	46
NAV steering cmds.	200	35	85	36	74
Display store update	200	36	95	37	97
Display keyset	200	37	96	38	98
Display status update	200	40	99	41	101

Figure 16.4
Aircraft controller schedulability results.

time to execute. This indicates that the system is still not close to saturation, although the total CPU utilization is high.

Notice also that preemption does not have a big impact on response times. Except for the most critical process, all others maintain their schedulability if a nonpreemptive scheduler is used. Although nonpreemptive scheduling causes weapon release to miss its deadline, the extra delay is small. If preemptive scheduling were expensive, reducing the CPU utilization slightly might make the complete system schedulable without changing the scheduler. By having such information the designer can easily assess the impact on various alternatives to improve the performance.

To see how the designer can use these results, we can analyze the response time for the display graphic subsystem. The period of this subsystem is 80 ms, and a shorter period might be desired to make motion look continuous. However, the response time of this process can be as high as 44 ms. Changing the period to 40 ms would make it miss its deadline. The designer may choose to decrease the period to 50 ms, for example. To test the effect of this change, the model can be analyzed again in order to check schedulability.

This kind of analysis can also be used to determine execution times for more complex sequences of events. For example, when a pilot presses the firing button, many subsystems

are involved in identifying and responding to this event. Analysis of the system using the algorithms in Section 16.4 shows that the minimum time between detecting that the fire button has been depressed and the end of execution of weapon release is 120 ms while the maximum time is 167 ms. By examining these times the designer is able to determine if the weapon system responds quickly enough to satisfy the aircraft requirements.

In this section we have shown how a system with complex timing constraints can be analyzed with a tool like VERUS. We have been able to determine the schedulability of the system and understand its behavior in detail. We have also been able to determine information about its behavior, such as the response time of the weapons subsystem. that might be difficult to obtain using other methods.

17 Continuous Real Time

In the previous chapter, we assumed that time is *discrete*. When time is modeled in this manner, possible clock values are nonnegative integers, and events can only occur at integer time values. This type of model is appropriate for *synchronous systems*, where all of the components are synchronized by a single global clock. The duration between successive clock ticks is chosen as the basic unit for measuring time. This model has been successfully used for reasoning about the correctness of synchronous hardware designs for many years.

Continuous time, on the other hand, is the natural model for *asynchronous systems*, because the separation of events can be arbitrarily small. This ability is desirable for representing causally independent events in an asynchronous system. Moreover, no assumptions need to be made about the speed of the environment when this model of time is assumed [3].

In order to model asynchronous systems using discrete time, it is necessary to discretize time by choosing some fixed time quantum so that the delay between any two events will be a multiple of this time quantum. This is difficult to do *a priori*, and may limit the accuracy with which systems can be modeled. Brzozowski and Seger [38] have shown, for example, that theoretically the reachability problem for asynchronous circuits with bounded delays cannot be solved correctly when time is assumed to be discrete. Also, the choice of a sufficiently small time quantum to model an asynchronous system accurately may blow up the state space so that verification is no longer feasible (this may be more of a problem for explicit state model checkers than for symbolic model checkers, however).

Although a number of different models of continuous time have been proposed [8, 99, 135, 170, 222, 235, 252, 253], the *timed automata model* of Alur, Courcoubetis, and Dill [8, 99] has become the standard. Certainly, most of the research on continuous-time model checking is based on this model. In this chapter we will discuss the properties of timed automata and explain the major techniques that have been developed for verifying them. Because so much research has been done in this area, we will restrict this brief survey to the *reachability* problem for such automata [3, 5]. Algorithms for CTL model checking [8], LTL model checking [7], and testing inclusion between timed omega automata [5, 9] have been proposed. Tools based on these algorithms have been developed and tested on realistic examples [10, 92]. The interested reader should consult the papers referenced earlier to learn more about these techniques.

17.1 Timed Automata

A *timed automaton* [8, 99] is a finite automaton augmented with a finite set of real-valued *clocks*. We assume that transitions are instantaneous. However, time can elapse when the automaton is in a state or *location*. When a transition occurs, some of the clocks may be reset to zero. At any instant, the reading of a clock is equal to the time that has elapsed

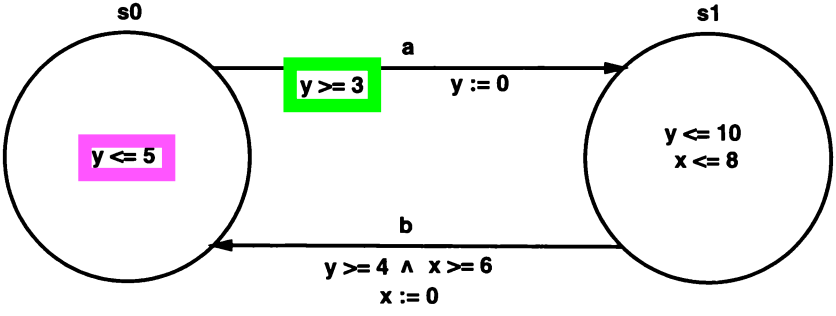


Figure 17.1
A simple timed automaton.

since the last time the clock was reset. We assume that time passes at the same rate for all clocks. In order to prevent pathological behaviors, we only consider automata that are *non-Zeno*, that is, only a finite number of transitions can happen within a finite amount of time.

A clock constraint, called a *guard*, is associated with each transition. The transition can be taken only if the current values of the clocks satisfy the clock constraint. A clock constraint is also associated with each location of the automaton. This constraint is called the *invariant* of the location. Time can elapse in the location only as long as the invariant of the location is true. An example of a timed automaton is shown in Figure 17.1. The automaton consists of two locations $s0$ and $s1$, two clocks x and y, an "a" transition from $s0$ to $s1$, and a "b" transition from $s1$ to $s0$. The automaton starts in location $s0$. It can remain in that location as long as the clock y is less than or equal to 5. As soon as the value of y is greater than or equal to 3, the automaton can make an "a" transition to location $s1$ and reset the clock y to 0. The automaton can remain in location $s1$ as long as y is less than or equal to 10 and x is less than or equal to 8. When y is at least 4 and x is at least 6, it can make a "b" transition back to location $s0$ and reset x.

The remainder of this section contains a formal semantics for timed automata in terms of infinite state transition graphs [3, 8]. We begin with a precise definition of clock constraints. Let X be a set of *clock variables*, ranging over the nonnegative real numbers $\mathbb{R}^+$. Define the set of *clock constraints* $\mathcal{C}(X)$ as follows:

- All inequalities of the form $x \prec c$ or $c \prec x$ are in $\mathcal{C}(X)$, where $\prec$ is either $<$ or $\leq$ and c is a nonnegative rational number.
- If φ_1 and φ_2 are in $\mathcal{C}(X)$, then $\varphi_1 \wedge \varphi_2$ is in $\mathcal{C}(X)$.

Note that if X contains k clocks, then each clock constraints is a *convex* subset of k-dimensional Euclidean space. Thus, if two points satisfy a clock constraint, then all of the points on the line segment connecting these points satisfy the clock constraint.

A *timed automaton* is a 6-tuple $A = (\Sigma, S, S_0, X, I, T)$ such that

- Σ is a finite *alphabet*.
- S is a finite set of *locations*.
- $S_0 \subseteq S$ is a set of *starting locations*.
- X is a set of clocks.
- $I : S \rightarrow \mathcal{C}(X)$ is a mapping from locations to clock constraints, called the *location invariant*.
- $T \subseteq S \times \Sigma \times \mathcal{C}(X) \times 2^X \times S$ is a set of transitions. The 5-tuple $\langle s, a, \varphi, \lambda, s' \rangle$ corresponds to a transition from location s to location s' labeled with a, a constraint φ that specifies when the transition is enabled, and a set of clocks $\lambda \subseteq X$ that are reset when the transition is executed.

We will require that time be allowed to progress to infinity, that is, at each location the upper bound imposed on the clocks be either infinity, or smaller than the maximum bound imposed by the invariant and by the transitions outgoing from the location. In other words, it is possible either to stay at a location forever, or the invariant will force the automaton to leave the location, and at that point at least one transition will be enabled. For timed automata, these constraints can be imposed syntactically.

A model for a timed automaton A is an infinite state transition graph $\mathcal{T}(A) = (\Sigma, Q, Q_0, R)$. Each state in Q is a pair (s, ν) where $s \in S$ is a location and $\nu : X \rightarrow \mathbb{R}^+$ is a *clock assignment*, mapping each clock to a nonnegative real value. The set of *initial states* Q_0 is given by $\{(s, \nu) | s \in S_0 \wedge \forall x \in X[\nu(x) = 0]\}$.

In order to define the state transition relation for $\mathcal{T}(A)$, we must first introduce some notation. For $\lambda \subseteq X$, define $\nu[\lambda := 0]$ to be the clock assignment that is the same as ν for clocks in $X - \lambda$ and maps the clocks in λ to 0. For $d \in \mathbb{R}$, define $\nu + d$ as the clock assignment that maps each clock $x \in X$ to $\nu(x) + d$. The clock assignment $\nu - d$ is defined in the same manner.

From the brief discussion in the introduction, we know that a timed automaton has two basic types of transitions:

- *Delay transitions* correspond to the elapsing of time while staying at some location. We write $(s, \nu) \xrightarrow{d} (s, \nu + d)$, where $d \in \mathbb{R}^+$, provided that for every $0 \leq e \leq d$, the invariant $I(s)$ holds for $\nu + e$.

- *Action transitions* correspond to the execution of a transition from T. We write $(s, \nu) \xrightarrow{a} (s', \nu')$, where $a \in \Sigma$, provided that there is a transition $\langle s, a, \varphi, \lambda, s' \rangle$ such that ν satisfies φ and $\nu' = \nu\,[\lambda := 0]$.

The transition relation R of $\mathcal{T}(A)$ is obtained by combining the delay and action transitions. We will write $(s, \nu)\ R\ (s', \nu')$ or $(s, \nu) \overset{a}{\Longrightarrow} (s', \nu')$ if there exists s'' and ν'' such that $(s, \nu) \xrightarrow{d} (s'', \nu'') \xrightarrow{a} (s', \nu')$ for some $d \in \mathbb{R}$.

In this chapter we will describe an algorithm for solving the *reachability problem* for $\mathcal{T}(A)$: Given a set of initial states Q_0, we show how to compute the set of all states $q \in Q$ that are reachable from Q_0 by transitions in R. This problem is nontrivial because $\mathcal{T}(A)$ has an infinite number of states. In order to accomplish this goal, it is necessary to use a finite representation for the infinite state space of $\mathcal{T}(A)$. Developing such representations is the main topic of the following sections.

17.2 Parallel Composition

Before we consider the reachability problem, we show how real-time systems can be modeled as parallel compositions of timed automata [3, 5]. We assume an interleaving or asynchronous semantics for this operation. Let $A_1 = (\Sigma_1, S_1, S_0^1, X_1, I_1, T_1)$ and $A_2 = (\Sigma_2, S_2, S_0^2, X_2, I_2, T_2)$ be two timed automata. Assume that the two automata have disjoint sets of clocks, that is, $X_1 \cap X_2 = \emptyset$. Then, the *parallel composition* of A_1 and A_2 is the timed automaton

$$A_1 \parallel A_2 = \langle \Sigma_1 \cup \Sigma_2,\ S_1 \times S_2,\ S_0^1 \times S_0^2,\ X_1 \cup X_2,\ I,\ T \rangle,$$

where $I(s_1, s_2) = I_1(s_1) \wedge I_2(s_2)$ and the edge relation T is given by the following rules:

1. For $a \in \Sigma_1 \cap \Sigma_2$, if $\langle s_1, a, \varphi_1, \lambda_1, s_1' \rangle \in T_1$ and $\langle s_2, a, \varphi_2, \lambda_2, s_2' \rangle \in T_2$, then T will contain the transition $\langle (s_1, s_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (s_1', s_2') \rangle$.
2. For $a \in \Sigma_1 - \Sigma_2$, if $\langle s, a, \varphi, \lambda, s' \rangle \in T_1$ and $t \in S_2$, then T will contain the transition $\langle (s, t), a, \varphi, \lambda, (s', t) \rangle$.
3. For $a \in \Sigma_2 - \Sigma_1$, if $\langle s, a, \varphi, \lambda, s' \rangle \in T_2$ and $t \in S_1$, then T will contain the transition $\langle (t, s), a, \varphi, \lambda, (t, s') \rangle$.

Thus, the locations of the parallel composition are pairs of locations from the component automata, and the invariant of such a location is the conjunction of the invariants of the component locations. There will be a transition in the parallel composition for each pair of transitions from the individual timed automata with the same action. The source location of the transition will be the composite location obtained from the source locations of the individual transitions. The target location will be the composite location obtained from

the target locations of the individual transitions. The guard will be the conjunction of the guards for the individual transitions, and the set of clocks that are reset will be the union of the sets that are reset by the individual transitions. If the action of a transition is only an action of one of the two processes, then there will be a transition in the parallel composition for each location of the other timed automaton. The source and target locations of these transitions will be obtained from the source and target locations of the original transition and the location from the other automaton. All of the other components of the transition will remain the same.

17.3 Modeling with Timed Automata

To illustrate how timed automata can be used to model real-time systems, we consider a simple manufacturing plant taken from Daws and Yovine [93]. The plant consists of a conveyor belt that moves from left to right, a processing station, and two robots that move boxes between the station and the belt as shown in Figure 17.2. The first robot (called the "D-Robot") takes a box from the station and *deposits* it on the left end of the belt. The

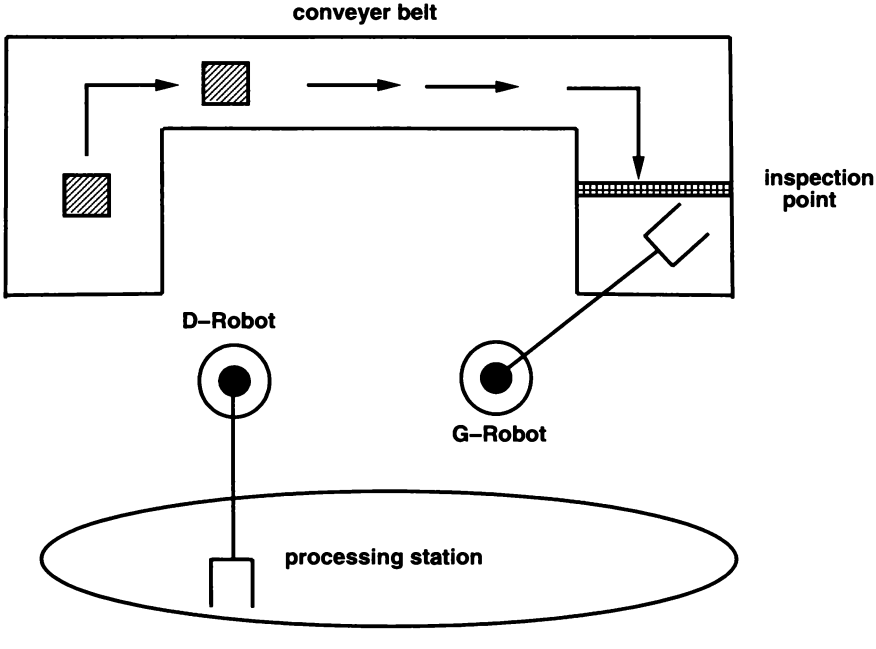


Figure 17.2
A manufacturing example.

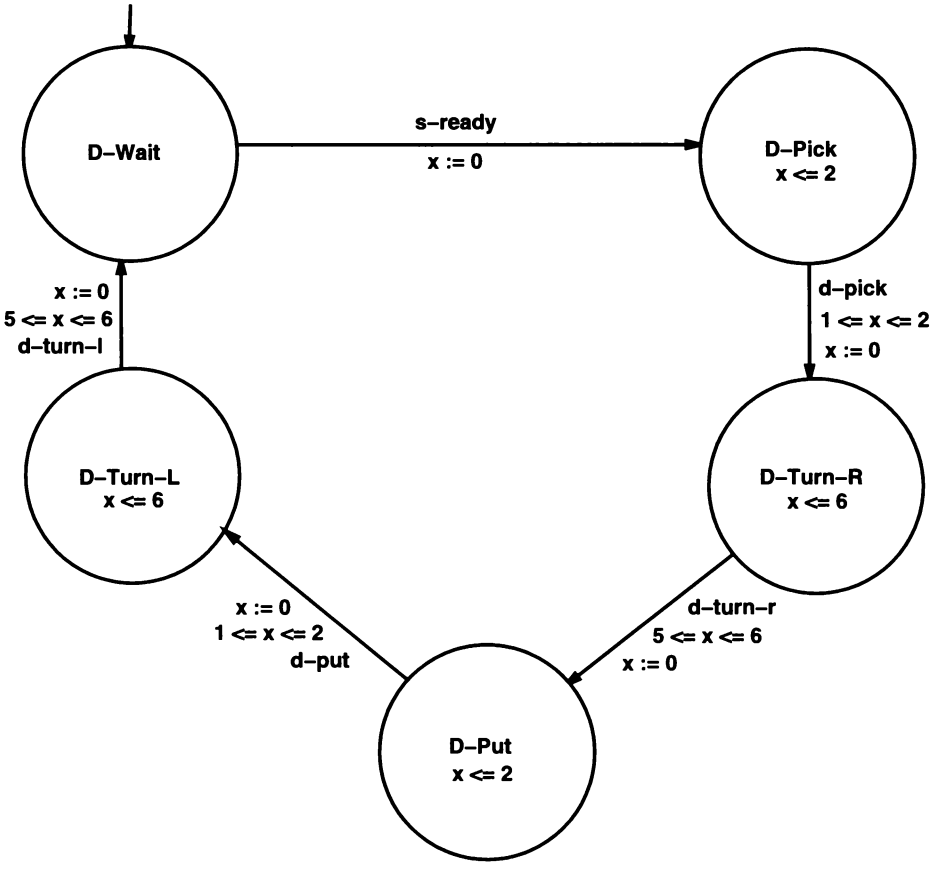


Figure 17.3
Timed automaton for D-Robot.

second robot (called the "G-Robot") *gets* a box from the right end of the belt and transfers it to the station where boxes are processed. Below, we describe each of these components in more detail.

The timed automaton for the D-Robot is shown in Figure 17.3. The robot waits by the station (in location D-Wait) until a box is ready (indicated by the action s-ready). Next, it picks the box up (D-Pick), turns right (D-Turn-R) and puts the box on the moving belt (D-Put). It then turns left (D-Turn-L) and returns to its initial position. Picking up the box or putting it down requires between one and two seconds. Turning left or right takes between five and six seconds.

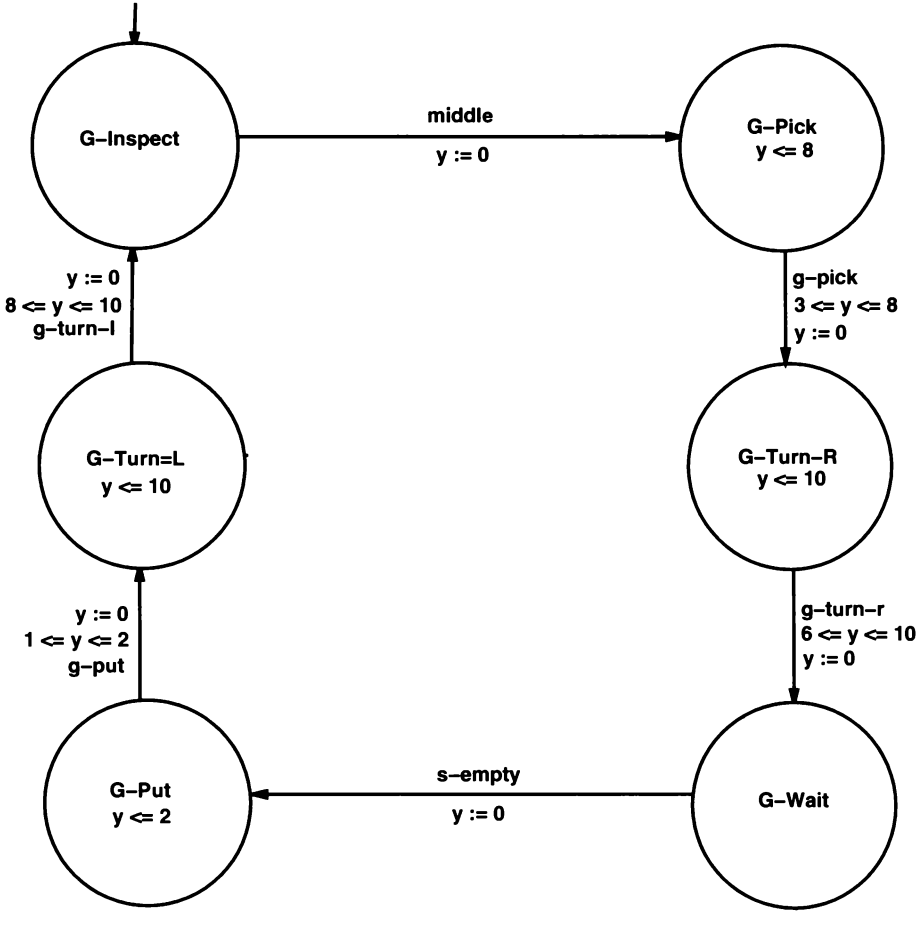


Figure 17.4
Timed automaton for G-Robot.

The timed automaton for the G-Robot is given in Figure 17.4. This robot waits (in location G-Inspect) at an inspection point near the right end of the belt until a box passes this point. The robot must pick up the box (G-Pick) before it falls off the end of the belt. Next, it turns right (G-Turn-R), waits for the station to finish processing the previous box (G-Wait), and then places the box at the station (G-Put). Finally, it turns left (G-Turn-L) back to the inspection point. It takes the robot between three and eight seconds to pick up the box and between six and ten seconds to turn right. It requires between one and two

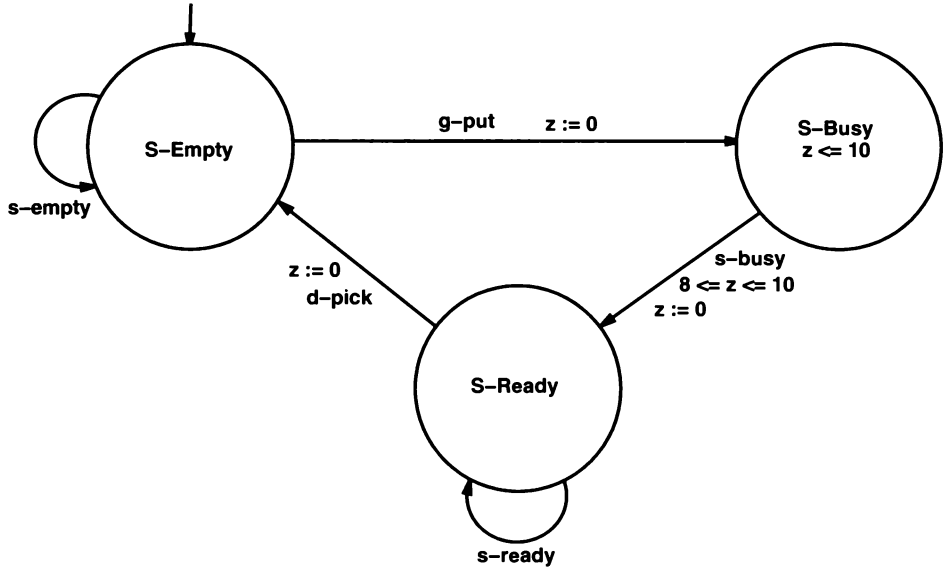


Figure 17.5
Time automaton for processing station.

seconds to place the box at the station and between eight and ten seconds to return to the inspection point.

The timed automaton for the processing station is shown in Figure 17.5. The station is initially empty (S-Empty). Once a box arrives at the station, it requires between eight and ten seconds to be processed. The box is then ready for the D-Robot to pick it up.

The timed automaton for the box is described in Figure 17.6. Initially the box is moving (B-Mov) from the left end of the belt to the inspection point. Once it passes the inspection point (B-Inspect), the box will fall off the belt (B-Fall) unless it is picked up by the G-Robot (B-on-G). In the latter case, the box is then placed at the station (B-on-S), picked up by the D-Robot (B-on-D) and put back on the left end of the belt. It takes between 133 and 134 seconds for the box to reach the inspection point from the left end of the belt. The box will fall off the belt if it is not picked up between twenty and twenty-one seconds after passing the inspection point.

The timed automaton for the system is the parallel composition of the four individual timed automata described above.

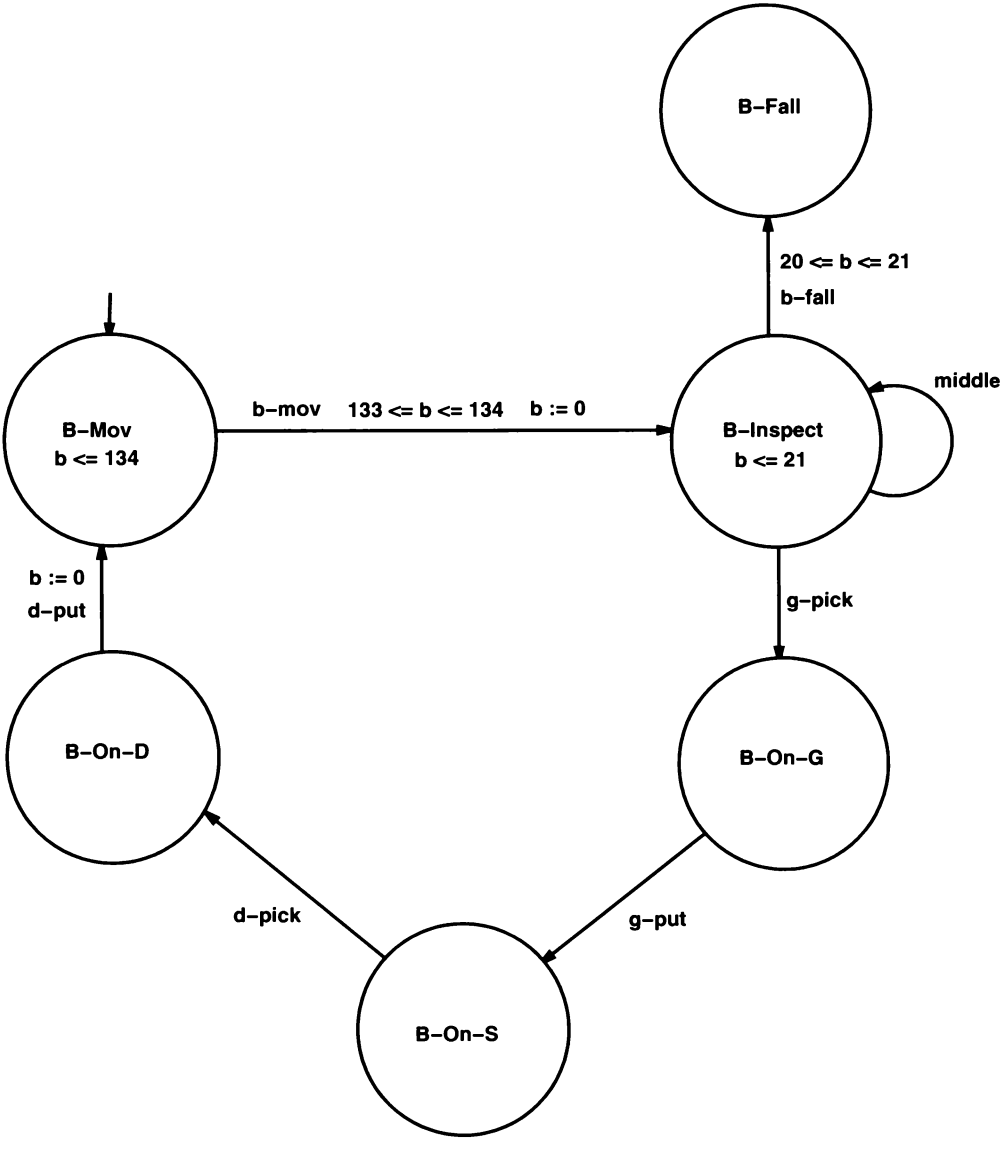


Figure 17.6
Timed automaton for box.

17.4 Clock Regions

In the definition of timed automata, we allowed the clock constraints that serve as the invariants of locations and the guards of transitions to contain arbitrary rational constants. We can multiply the constants in each clock constraint by the least common multiple m of the denominators of all the constants that appear in all of the constraints [3]. This converts all of the constants to integers. The value of a clock can still be an arbitrary nonnegative real number. Note that applying this transformation can change the clock assignments in the set of reachable states of $\mathcal{T}(A)$. Fortunately, this does not cause a major problem. The reachable states of the original automaton can be obtained from the locations of the transformed automaton by applying the inverse transformation, that is, dividing each clock value by m.

The largest constant in the transformed automaton is the product of m and the largest constant in the original automaton. Thus, the transformation at worst results in a quadratic blowup in the length of the encodings of the clock constraints [3]. This increase in complexity is acceptable, since the transformation simplifies certain operations on clock constraints that will be needed later in the chapter. We will apply this transformation uniformly to all of the clock constraints that appear in the timed automata that we study. Consequently, in the future we can assume without loss of generality that all constants in clock constraints that we encounter are integers.

In order to obtain a finite representation for the infinite state space of a timed automaton, we define *clock regions* [7, 8], which represent sets of clock assignments. If two states, which correspond to the same location of the timed automaton A, agree on the integral parts of all clock values and also on the ordering of the fractional parts of all the clocks, then the states will behave in a similar manner. The integral parts of the clock values determine whether a clock constraint in the invariant of a location or in the guard of a transition is satisfied or not. The ordering of the fractional parts of the clock values determines which clock will change its integral part first. This is because clock constraints can involve only integers, and all clocks increase at the same rate.

For example, let A be a timed automaton with two clocks x_1 and x_2. Let s be a location in A with an outgoing transition e to some other location. Consider two states (s, ν) and (s, ν') in $\mathcal{T}(A)$ that correspond to location s. Suppose that $\nu(x_1) = 5.3$, $\nu(x_2) = 7.5$, $\nu'(x_1) = 5.5$, and $\nu'(x_2) = 7.9$. Assume that the guard φ associated with e is $x_1 \geq 8 \wedge x_2 \geq 10$. It is easy to see that if (s, ν) eventually satisfies the guard, then so will (s, ν').

The value of a clock can get arbitrarily large; however, if the clock is never compared to a constant greater than c, then the value of the clock will have no effect on the computation of A once it exceeds c. Suppose, for instance, that the clock x is never compared to a constant greater than 100 in the invariant associated with a location or in the guard of a transition.

Then, based on the behavior of A, it is impossible to distinguish between x having the value 101 and x having the value 1001.

Alur, Courcoubetis, and Dill [7, 8] show how to formalize this reasoning. For each clock $x \in X$, let c_x be the largest constant that x is compared with in the invariant of any location or in the guard of any transition. For $t \in \mathbb{R}^+$, let $fr(t)$ be the fractional part of t, and let $\lfloor t \rfloor$ be the integral part of t. Thus, $t = \lfloor t \rfloor + fr(t)$. We define an equivalence relation $\cong$ on the set of possible clock assignments as follows: Let ν and ν' be two clock assignments. Then, $\nu \cong \nu'$ if and only if three conditions are satisfied:

1. For all $x \in X$ either $\nu(x) \geq c_x$ and $\nu'(x) \geq c_x$ or

$\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$.

2. For all $x, y \in X$ such that $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$,

$fr(\nu(x)) \leq fr(\nu(y))$ if and only if $fr(\nu'(x)) \leq fr(\nu'(y))$.

3. For all $x \in X$ such that $\nu(x) \leq c_x$,

$fr(\nu(x)) = 0$ if and only if $fr(\nu'(x)) = 0$.

It is easy to see that $\cong$ does indeed define an equivalence relation. The equivalence classes of $\cong$ are called *regions* [7, 8]. We will write $[\nu]$ to denote the region which contains the clock assignment ν. Each region can be represented by specifying

1. for every clock $x \in X$, one clock constraint from the set

$\{x = c \mid c = 0, \ldots, c_x\} \cup \{c - 1 < x < c \mid c = 1, \ldots, c_x\} \cup \{x > c_x\}$,

2. for every pair of clocks $x, y \in X$ such that $c - 1 < x < c$ and $d - 1 < y < d$ are clock constraints in the first condition, whether $fr(x)$ is less than, equal to, or greater than $fr(y)$.

Figure 17.7, which is taken from [8], shows the clock regions for a timed automaton with two clocks x and y where $c_x = 2$ and $c_y = 1$. In this example, there are a total of 28 regions: 6 corner points (e.g. $[(1, 0)]$), 14 open line segments (e.g., $[1 < x < 2 \wedge y = x - 1]$), and 8 open regions (e.g., $[1 < x < 2 \wedge 0 < y < x - 1]$).

We will use this observation to show that $\cong$ has finite index and, consequently, that the number of regions is finite. Our proof of this fact is based on the proof given in [8].

LEMMA 43 The number of equivalence classes (i.e., clock regions) that $\cong$ induces on $\mathcal{C}(X)$ is bounded by

$$|X|! \cdot 2^{|X|} \cdot \prod_{x \in X} (2c_x + 2).$$

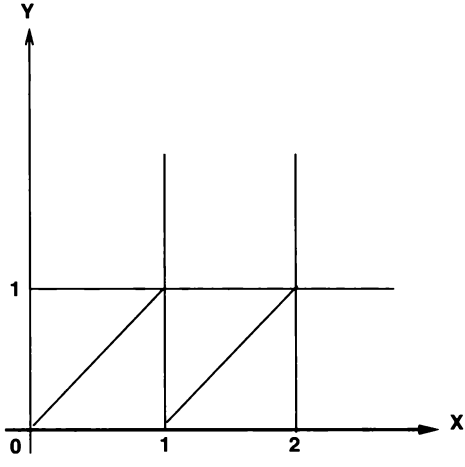


Figure 17.7
Clock region example.

Proof An equivalence class $[\nu]$ of $\cong$ can be described by a triple of arrays $\langle \alpha, \beta, \gamma \rangle$ in the following manner: For each clock $x \in X$, the array α tells which of the intervals

$$\{[0, 0], (0, 1), [1, 1], \ldots, (c_x - 1, c_x), [c_x, c_x], (c_x, \infty)\}$$

contains the value $\nu(x)$. Thus, the array α represents the clock assignment ν if and only if for each clock $x \in X$, $\nu(x) \in \alpha(x)$. The number of ways to choose α is $\prod_{x \in X}(2c_x + 2)$.

Let X_α be the set of clocks x such that $\alpha(x)$ has the form $(i, i+1)$ for some $i \leq c_x$. Thus, X_α is the set of clocks with nonzero fractional part. The array $\beta : X_\alpha \to \{1, \ldots, |X_\alpha|\}$ is a permutation of X_α, which gives the ordering of the fractional parts of the clocks in X_α with respect to $\leq$. Thus, the array β represents a clock assignment ν if and only if for each pair $x, y \in X_\alpha$, if $\beta(x) < \beta(y)$ then $fr(\nu(x)) \leq fr(\nu(y))$. For a given α, the number of ways to choose β is bounded by $|X_\alpha|!$ which is bounded by $|X|!$.

The third component γ is a boolean array indexed by X_α that is used to specify which clocks in X_α have the same fractional part. For each clock x, $\gamma(x)$ tells whether or not the fractional part of $\nu(x)$ equals the fractional part of its predecessor in the array β. Thus, the array γ represents a clock assignment ν if and only if for each $x \in X_\alpha$, $\gamma(x)$ equals 0 exactly when there is a clock $y \in X_\alpha$ such that $\beta(y) = \beta(x) + 1$ and $fr(\nu(x))$ equals $fr(\nu(y))$. The number of ways of choosing γ is bounded by the number of boolean arrays over X_α, which is bounded by $2^{|X|}$.

Hence, α encodes the integral parts of the clock assignments, and β together with γ encodes the ordering of their fractional parts. It is easy to see that the sets represented

by triples are equivalence classes of $\cong$ and that every equivalence class is represented by some triple. The bound given in the statement of the lemma is the product of the bounds associated with α, β, and γ. This completes the proof of the lemma. □

The following properties of the equivalence relation $\cong$ are used later in this chapter.

LEMMA 44 Let ν_1 and ν_2 be two clock assignments, let φ be a clock constraint, and let $\lambda \subseteq X$ be a set of clocks.

1. If $\nu_1 \cong \nu_2$ and t is a nonnegative integer, then $\nu_1 + t \cong \nu_2 + t$.
2. If $\nu_1 \cong \nu_2$, then $\forall t_1 \in \mathbb{R}^+ \; \exists t_2 \in \mathbb{R}^+ \; [\nu_1 + t_1 \cong \nu_2 + t_2]$.
3. If $\nu_1 \cong \nu_2$, then ν_1 satisfies φ if and only if ν_2 satisfies φ.
4. If $\nu_1 \cong \nu_2$, then $\nu_1 \, [\lambda := 0] \cong \nu_2 \, [\lambda := 0]$.

Note that the first property may not hold if t is not an integer. For example, $(.2, .8) \cong (.1, .2)$, but $(.2, .8) + .3$ is not equivalent to $(.1, .2) + .3$. All of the properties except the second are straightforward to prove and will be left to the reader. A proof of the second property is sketched below. The proof is not difficult, but it is somewhat tedious. It can be safely skipped when this chapter is read for the first time.

Proof Assume that $\nu_1 \cong \nu_2$. We can assume that $t_1 > 0$ because, otherwise, we can simply choose $t_2 = 0$. Let $X = \{x_1, x_2, \ldots, x_n\}$. We can treat ν_1 as a vector $\nu_1 = \langle a_1, \ldots, a_n \rangle$, where a_i is the value of clock x_i in ν_1. Similarly, we let $\nu_2 = \langle b_1, \ldots, b_n \rangle$. Since corresponding clocks have the same integer part, we can assume without loss of generality that $0 \leq a_i < 1$ and $0 \leq b_i < 1$. Also, assume that the clock values are sorted into increasing order so that $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$.

Case 1 Assume that the largest element in $\nu_1 + t_1$ is less than or equal to 1. This case is trivial. We can easily choose t_2 so that $\nu_1 + t_1 \cong \nu_2 + t_2$.

Case 2 Assume that $0 \leq t_1 < 1$. Let the first element of $\nu_1 + t_1$ that is greater than or equal to 1 be $a_k + t_1$. Choose ϵ so that $\epsilon = 0$ if $a_k + t_1 = 1$ and so that $0 < \epsilon < b_k - b_{k-1}$ if $a_k + t_1 > 1$. Note that $b_{k-1} < b_k$. If $b_k = b_{k-1}$, then $a_k = a_{k-1}$ and $a_k + t_1$ is not the first element of $\nu_1 + t_1$ that is greater than or equal to 1. We will show that $\nu_1 + t_1 \cong \nu_2 + (1 + \epsilon - b_k)$. In order to show this we will split the vectors into two parts. Let

$$L_1 = \langle a_1 + t_1, \ldots, a_{k-1} + t_1 \rangle, \text{ and}$$

$$L_2 = \langle b_1 + (1 + \epsilon - b_k), \ldots, b_{k-1} + (1 + \epsilon - b_k) \rangle.$$

In each case it is straightforward to show that

1. all of the elements are positive,
2. the elements are sorted in increasing order, and
3. all of the elements are less than 1.

Because of these conditions it is easy to see that $L_1 \cong L_2$. Similarly, let

$R_1 = \langle a_k + t_1, \ldots, a_n + t_1 \rangle$, and

$R_2 = \langle b_k + (1 + \epsilon - b_k), \ldots, b_n + (1 + \epsilon - b_k) \rangle$.

All of the elements in R_1 and R_2 are greater than or equal to 1. The fractional parts are given by $R_1 - 1$ and $R_2 - 1$, respectively. For these vectors it is straightforward to show that

1. all of the elements are nonnegative,
2. the elements are sorted in increasing order, and
3. all of the elements are less than 1.

Moreover, an element in one vector is 0 if and only if the corresponding element in the other vector is 0. Thus, $R_1 - 1 \cong R_2 - 1$. It follows immediately that $R_1 \cong R_2$.

It is not difficult to see that the fractional parts of R_2 precede the fractional parts of L_2. Let $i \geq k$ and $j < k$. Then

$b_i + (1 + \epsilon - b_k) - 1 \leq b_j + (1 + \epsilon - b_k)$.

is equivalent to $b_i - b_j \leq 1$, which is obviously true. The same relationship holds for the fractional parts of R_1 and L_1, that is,

$a_i + t_1 - 1 \leq a_j + t_1$.

Hence, we obtain $R_1 \cdot L_1 \cong R_2 \cdot L_2$, where "$\cdot$" is concatenation of vectors. This shows that for all t_1 with $0 \leq t_1 < 1$, there exists a t_2 such that $\nu_1 + t_1 \cong \nu_2 + t_2$ and completes the proof of Case 2.

Case 3 Finally, suppose that $t_1 \geq 1$. Let $t_1' = t_1 - \lfloor t_1 \rfloor$, so that $0 \leq t_1' < 1$. Find t_2' such that $v_1 + t_1' \cong v_2 + t_2'$. Then

$\nu_1 + t_1' + \lfloor t_1 \rfloor \cong \nu_2 + t_2' + \lfloor t_1 \rfloor$.

If we choose $t_2 = t_2' + \lfloor t_1 \rfloor$, then we have $\nu_1 + t_1 \cong \nu_2 + t_2$ as required. This completes the proof of the second property. □

The equivalence relation $\cong$ over clock assignments can be extended to an equivalence relation over the state space of $\mathcal{T}(A)$ by requiring that equivalent states have identical

locations and equivalent clock assignments: $(s, \nu) \cong (s', \nu')$ if and only if $s = s'$ and $\nu \cong \nu'$. The key property of the equivalence relation $\cong$ is given by the following lemma [5]:

LEMMA 45 If $\nu_1 \cong \nu_2$ and $(s, \nu_1) \stackrel{a}{\Longrightarrow} (s', \nu'_1)$, then there exists a clock assignment ν'_2 such that $\nu'_1 \cong \nu'_2$ and $(s, \nu_2) \stackrel{a}{\Longrightarrow} (s', \nu'_2)$.

Proof Assume that $\nu_1 \cong \nu_2$ and $(s, \nu_1) \stackrel{a}{\Longrightarrow} (s', \nu'_1)$. The transition $\langle s, a, \varphi, \lambda, s' \rangle$ that takes state (s, ν_1) to state (s', ν'_1) corresponds to two transitions of the timed automaton:

- a delay transition $(s, \nu_1) \xrightarrow{d_1} (s, \nu_1 + d_1)$, for some $d_1 \geq 0$, and
- an action transition $(s, \nu_1 + d_1) \xrightarrow{a} (s', \nu'_1)$ such that $\nu_1 + d_1$ satisfies φ and $\nu'_1 = (\nu_1 + d_1)[\lambda := 0]$.

Since $\nu_1 \cong \nu_2$ and ν_1 satisfies $I(s)$, ν_2 also satisfies $I(s)$. Furthermore, there exists $d_2 \geq 0$ such that $\nu_1 + d_1 \cong \nu_2 + d_2$. Since $\nu_1 + d_1$ satisfies $I(s)$, $\nu_2 + d_2$ also satisfies $I(s)$. Because the clock constraint $I(s)$ is convex and is satisfied by both ν_2 and $\nu_2 + d_2$, $I(s)$ must be satisfied by $\nu_2 + e$ for all e such that $0 \leq e \leq d_2$. Consequently, the delay transition $(s, \nu_2) \xrightarrow{d_2} (s, \nu_2 + d_2)$ is legal.

Since $\nu_1 + d_1 \cong \nu_2 + d_2$, both $\nu_1 + d_1$ and $\nu_2 + d_2$ must satisfy the clock constraint for the guard φ. Thus, the transition $\langle s, a, \varphi, \lambda, s' \rangle$ must also be enabled in the state $(s, \nu_2 + d_2)$. Let $\nu'_2 = (\nu_2 + d_2)[\lambda := 0]$. Then ν'_2 is equivalent to ν'_1. Hence, there is an action transition $(s, \nu_2 + d_2) \xrightarrow{a} (s', \nu'_2)$. Combining the delay transition with the action transition, we get $(s, \nu_2) \stackrel{a}{\Longrightarrow} (s', \nu'_2)$ as required. □

As a result of the lemma, we can construct a finite state transition graph that is bisimulation equivalent to the infinite state transition graph $\mathcal{T}(A)$. The finite state transition graph is called the *region graph* of A [7, 8] and is denoted by $\mathcal{R}(A)$. A *region* is a pair $(s, [\nu])$. Since $\cong$ has a finite index, there are only a finite number of regions. The states of the region graph are the regions of A. The construction of $\mathcal{R}(A)$ will have the property that whenever (s, ν) is a state of $\mathcal{T}(A)$, the region $(s, [\nu])$ will be a state of $\mathcal{R}(A)$. The initial states of the region graph have the form $(s_0, [\nu_0])$ where s_0 is an initial state of A and ν_0 is a clock assignment that assigns 0 to every clock. The transition relation of $\mathcal{R}(A)$ is defined so that bisimulation equivalence is guaranteed. There will be a transition labeled with a from the region $(s, [\nu])$ to the region $(s', [\nu'])$ if and only if there are assignments $\omega \in [\nu]$ and $\omega' \in [\nu']$ such that (s, ω) can make a transition to (s', ω').

We summarize the construction of the region graph $\mathcal{R}(A)$ below. Let $A = (\Sigma, S, S_0, X, I, T)$ be a timed automaton. Then,

- The states of $\mathcal{R}(A)$ have the form $(s, [\nu])$ where $s \in S$ and $[\nu]$ is a clock region.
- The initial states have the form $(s_0, [\nu])$ where $s_0 \in S_0$ and $\nu(x) = 0$ for all $x \in X$.

- $\mathcal{R}(A)$ has a transition $((s, [\nu]), a, (s', [\nu']))$ if and only if $(s, \omega) \xRightarrow{a} (s', \omega')$ for some $\omega \in [\nu]$ and some $\omega' \in [\nu']$.

We can use Lemma 45 to prove bisimulation equivalence.

THEOREM 31 The state transition graph $\mathcal{T}(A)$ and the region graph $\mathcal{R}(A)$ are bisimilar as transition systems.

Proof We will show that $\mathcal{T}(A)$ and $\mathcal{R}(A)$ are bisimilar. Define the bisimulation relation B by $(s, \nu)B(s, [\nu])$. It is easy to see that the initial state (s_0, ν_0) corresponds to the state $(s_0, [\nu_0])$. Next, we show that for each transition of $\mathcal{T}(A)$, there is a corresponding transition of $\mathcal{R}(\mathcal{A})$, and vice versa. Suppose first that $(s, \nu)B(s, [\nu])$ and that $(s, \nu) \xRightarrow{a} (s', \nu')$. It follows immediately that $(s, [\nu]) \xRightarrow{a} (s', [\nu'])$ and that $(s', \nu')B(s', [\nu'])$. Suppose, on the other hand, that $(s, \nu)B(s, [\nu])$ and that $(s, [\nu]) \xRightarrow{a} (s', [\nu'])$. Then, there exists $\omega \cong \nu$ and $\omega' \cong \nu'$ such that $(s, \omega) \xRightarrow{a} (s', \omega')$. Since $(s, \omega) \cong (s, \nu)$, by Lemma 45 there exists (s', ν'') such that $(s', \omega') \cong (s', \nu'')$ and $(s, \nu) \xRightarrow{a} (s', \nu'')$. Hence, $\nu'' \cong \omega' \cong \nu'$, so $[\nu''] = [\nu']$. By the definition of B, $(s', \nu'')B(s', [\nu''])$, it follows that $(s', \nu'')B(s', [\nu'])$. □

17.5 Clock Zones

An alternative way to obtain a finite representation for the infinite state space $\mathcal{T}(A)$ is to define *clock zones* [3], which also represent sets of clock assignments. A clock zone is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. We allow inequalities of the following types:

$$x \prec c, \; c \prec x, \; x - y \prec c,$$

where $\prec$ is $<$ or $\leq$.

By introducing a special clock x_0 that is always 0, it is possible to obtain a more uniform notation for clock zones. Since the value of a clock is always nonnegative, we will assume that constraints involving only one clock have the form

$$-c_{0,i} \prec x_i \prec c_{i,0},$$

where $-c_{0,i}$ and $c_{i,0}$ are both nonnegative. Using the special clock x_0, we will replace this constraint by the conjunction of two inequalities

$$x_0 - x_i \prec c_{0,i} \wedge x_i - x_0 \prec c_{i,0}.$$

Thus, the general form of a clock zone is

$$x_0 = 0 \wedge \bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j}.$$

The following operations will be used to construct more complicated clock zones from simpler ones [3]. Let φ be a clock zone. If $\lambda \subseteq X$ is a set of clocks, then define $\varphi[\lambda := 0]$ to be the set of all clock assignments $\nu\ [\lambda := 0]$ where $\nu \in \varphi$. If $d \in \mathbb{R}^+$, then we define $\varphi + d$ to be the set of all clock assignments $\nu + d$ where $\nu \in \varphi$. The set $\varphi - d$ is defined similarly.

Let φ be a clock zone expressed in terms of the clocks in X. The conjunction φ will represent a set of assignments to the clocks in X. If X contains k elements, then φ will be a convex subset of k-dimensional Euclidean space. The following lemma shows that the projection of a clock zone onto a lower dimensional subspace is also a clock zone.

LEMMA 46 If φ is a clock zone with free clock variable x, then $\exists x[\varphi]$ is also a clock zone.

This lemma turns out to be quite valuable in working with clock zones and will be proved at the end of the section.

Note that the assignment of values to the clocks in an initial state of timed automaton A is easily expressed as a clock zone since $\nu\ (x) = 0$ for every clock $x \in X$. Moreover, every clock constraint used in the invariant of an automaton location or in the guard of a transition is a clock zone. Because of this observation, clock zones can be used as the basis for various state reachability analysis algorithms for timed automata. These algorithms are usually expressed in terms of three operations on clock zones [3].

Intersection

If φ and ψ are two clock zones, then the intersection $\varphi \wedge \psi$ is a clock zone. This is easy to see. Because φ and ψ are clock zones, they can be expressed as conjunctions of clock constraints. Hence, $\varphi \wedge \psi$ is also a conjunction of clock constraints and, therefore, a clock zone.

Clock Reset

If φ is a clock zone and λ is a set of clocks, then $\varphi[\lambda := 0]$ is a clock zone. We will show that this is true when λ contains a single clock x. In this case, $\varphi[x := 0]$ is equivalent to $\exists x[\varphi \wedge x = 0]$, and the result follows immediately by Lemma 46. The result can easily be extended to sets with more than one clock by induction.

Elapsing of Time

We illustrate this operation first with a geometric example (see Figure 17.8). The triangular shaped area represents a simple clock zone φ. The area above the triangle φ is unbounded, and its sides (the dashed lines) make 45° angles with the horizontal axis. The triangle and

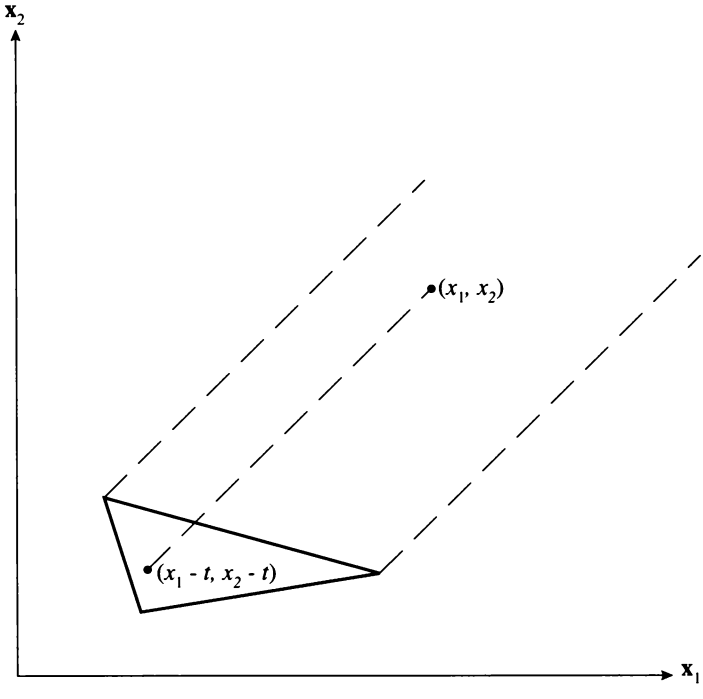


Figure 17.8
The clock zones φ and $\varphi^{\Uparrow}$.

the area above it represent the clock assignments that can be reached by time elapsing from an assignment in φ. This region is denoted by $\varphi^{\Uparrow}$.

Formally, if φ is a clock zone, then a clock assignment ν will be an element of $\varphi^{\Uparrow}$, if ν satisfies the formula $\exists t \geq 0[(\nu - t) \in \varphi]$ or, equivalently, $\exists t \geq 0[\nu \in (\varphi + t)]$. This region is a clock zone. In order to demonstrate this, we assume that t is a new clock and show that $\varphi + t$ is a clock zone that depends on the clocks in X and on t. We consider three types of inequalities:

1. $-c_{0,i} \prec x_i$: This inequality will become $-c_{0,i} \prec x_i - t$, which can be rewritten as $t - x_i \prec c_{0,i}$.
2. $x_i \prec c_{i,0}$: This inequality will become $x_i - t \prec c_{i,0}$, which is already in the appropriate form.
3. $x_i - x_j \prec c_{i,j}$: This inequality will become

$$(x_i - t) - (x_j - t) \prec c_{i,j}.$$

Because the two occurrences of the variable t cancel each other out, this inequality also has the appropriate form.

Since $\varphi + t$ is a clock zone, we can use Lemma 46 to show that $\varphi^{\Uparrow} = \exists t \geq 0[\varphi + t]$ is a clock zone that depends on X.

In principle, the three operations on clock zones described above can be used to construct a finite representation of the transition graph $\mathcal{T}(A)$ corresponding to a timed automaton. In the next section we will describe how this algorithm can be implemented efficiently by using *difference bound matrices* [3, 99]. In this section states are represented by *zones* [3]. A zone is a pair (s, φ) where s is a location of the timed automaton and φ is a clock zone. Consider a timed automaton A with transition $e = (s, a, \psi, \lambda, s')$. Assume that the current zone is (s, φ). Thus, s is a location of A, and φ is a clock zone. The clock zone $succ(\varphi, e)$ will denote the set of clock assignments ν' such that for some $\nu \in \varphi$, the state (s', ν') can be reached from the state (s, ν) by letting time elapse and the executing the transition e. The pair $(s', succ(\varphi, e))$ will represent the set of successors of (s, φ) under the transition e. The clock zone $succ(\varphi, e)$ is obtained by the following steps [3]:

1. Intersect φ with the invariant of location s to find the set of possible clock assignments for the current state.
2. Let time elapse in location s using the operator $\Uparrow$ described above.
3. Take the intersection with the invariant of location s again to find the set of clock assignments that still satisfy the invariant.
4. Take the intersection with the guard ψ of the transition e to find the clock assignments that are permitted by the transition.
5. Set all of the clocks in λ that are reset by the transition to 0.

Combining all of the above steps into one formula, we obtain

$$succ(\varphi, e) = ((\varphi \wedge I(s))^{\Uparrow} \wedge I(s) \wedge \psi)[\lambda := 0])$$

Because clock zones are closed under the operations of intersection, elapsing of time, and resetting of clocks, the set $succ(\varphi, e)$ is also a clock zone.

Finally, we describe how to construct a transition system for a timed automaton A. The transition system is called the *zone graph* and is denoted by $Z(A)$. The states of $Z(A)$ are the zones of A. If s is an initial location of A, then $(s, [X := 0])$ will be an initial state of $Z(A)$. There will be a transition from the zone (s, φ) in $Z(A)$ to the zone $(s', succ(\varphi, e))$

in $Z(A)$ labeled with the action a for each transition of the form $e = (s, a, \psi, \lambda, s')$ of the timed automaton A. Because each step in the construction of the zone graph is effective, this gives an algorithm for determining state reachability in the state transition graph $\mathcal{T}(A)$. In the next section we will show how to make this construction more efficient.

Before concluding this section, we will prove Lemma 46, which was stated without proof earlier in this section. The proof is not difficult, but can be safely skipped when reading the section for the first time.

Proof Assume that the clock zone φ is given by

$$x_0 = 0 \wedge \bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j},$$

where each instance of $\prec$ represents either $<$ or $\le$. Without loss of generality, we will prove that $\exists x_n[\varphi]$ is a clock zone when $n > 0$. In particular, we will show that $\exists x_n[\varphi]$ is given by

$$x_0 = 0 \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}.$$

In describing the proof we will normally omit writing the constraint $x_0 = 0$ because this equality is part of every formula and quantification over x_0 is not allowed. We will first prove that if an assignment to the variables $x_0, \ldots, x_{n-1}$ satisfies

$$\exists x_n \left[\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j} \right],$$

then it also satisfies

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}.$$

In order to see why this is true, we rewrite

$$\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j}$$

as

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i < n} x_i - x_n \prec c_{i,n} \wedge \bigwedge_{0 \le j < n} x_n - x_j \prec c_{n,j}$$

and consider the following chain of implications:

$$\bigwedge_{0 \le i < n} x_i - x_n \prec c_{i,n} \wedge \bigwedge_{0 \le j < n} x_n - x_j \prec c_{n,j}$$

$$\Rightarrow \bigwedge_{0 \le i \ne j < n} x_i - x_n \prec c_{i,n} \wedge x_n - x_j \prec c_{n,j}$$

$$\Rightarrow \bigwedge_{0 \le i \ne j < n} (x_i - x_n) + (x_n - x_j) \prec c_{i,n} + c_{n,j}$$

$$\Rightarrow \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}.$$

Because the last formula does not contain x_n, we see that

$$\exists x_n \left[\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j} \right]$$

implies

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}.$$

Next, we must show the reverse implication. We show that any assignment to $x_0, \ldots,$ x_{n-1} that makes

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}.$$

true will also make

$$\exists x_n \left[\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j} \right]$$

true. In other words, we must find a nonnegative value for the variable x_n so that

$$\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j}$$

is true. If

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}$$

is true, then

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}$$

will also be true. This formula can be rewritten as

$$\bigwedge_{0 \le i \ne j < n} x_i - c_{i,n} \prec x_j + c_{n,j}.$$

It follows that

$$\operatorname*{Max}_{0 \le i < n} (x_i - c_{i,n}) \prec \operatorname*{Min}_{0 \le j < n} (x_j + c_{n,j}).$$

We will choose the value for x_n so that

$$\operatorname*{Max}_{0 \le i < n} (x_i - c_{i,n}) \prec x_n \prec \operatorname*{Min}_{0 \le j < n} (x_j + c_{n,j}).$$

In particular, we will have $0 - c_{0,n} \prec x_n \prec 0 + c_{n,0}$. Since $-c_{0,n}$ and $c_{n,0}$ are both nonnegative, the value of x_n will also be nonnegative as required. It follows that

$$\bigwedge_{0 \le i \ne j < n} x_i - c_{i,n} \prec x_n \prec x_j + c_{n,j}$$

is also true. We can rewrite this formula as

$$\bigwedge_{0 \le i < n} x_i - c_{i,n} \prec x_n \wedge \bigwedge_{0 \le j < n} x_n \prec x_j + c_{n,j}.$$

Rearranging terms, we get

$$\bigwedge_{0 \le i < n} x_i - x_n \prec c_{i,n} \wedge \bigwedge_{0 \le j < n} x_n - x_j \prec c_{n,j}.$$

Consequently, if

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,n} + c_{n,j}$$

is true, then

$$\bigwedge_{0 \le i \ne j < n} x_i - x_j \prec c_{i,j} \wedge \bigwedge_{0 \le i < n} x_i - x_n \prec c_{i,n} \wedge \bigwedge_{0 \le j < n} x_n - x_j \prec c_{n,j}$$

will also be true. The last formula reduces to

$$\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j}.$$

This shows that

$$\exists x_n \left[\bigwedge_{0 \le i \ne j \le n} x_i - x_j \prec c_{i,j} \right]$$

is also true and completes the proof of the second half of the lemma. □

17.6 Difference Bound Matrices

A clock zone can be represented by a *difference bound matrix* as described by Dill in [99]. This matrix is indexed by the clocks in X together with a special clock x_0 whose value is always 0. This clock plays exactly the same role as the clock x_0 in the previous section. Each entry $\mathcal{D}_{i,j}$ in the matrix $\mathcal{D}$ has the form $(d_{i,j}, \prec_{i,j})$ and represents the inequality $x_i - x_j \prec d_{i,j}$, where $\prec_{i,j}$ is either $<$ or $\le$, or $(\infty, <)$, if no such bound is known. Because the variable x_0 is always 0, it can be used for expressing constraints that only involve a single variable. Thus, $\mathcal{D}_{j,0} = (d_{j,0}, \prec)$, means that we have the constraint $x_j \prec d_{j,0}$. Likewise, $\mathcal{D}_{0,j} = (d_{0,j}, \prec)$, means that we have the constraint $0 - x_j \prec d_{0,j}$ or $-d_{0,j} \prec x_j$.

To illustrate the use of difference bound matrices, consider the clock zone given by the formula

$$x_1 - x_2 < 2 \ \wedge \ 0 < x_2 \le 2 \ \wedge \ 1 \le x_1.$$

In this case we obtain the matrix D shown below:

	0	1	2
0	$(0, \le)$	$(-1, \le)$	$(0, <)$
1	$(\infty, <)$	$(0, \le)$	$(2, <)$
2	$(2, \le)$	$(\infty, <)$	$(0, \le)$

The representation of a clock zone by a difference bound matrix is not unique. In the above example, there are some implied constraints that are not reflected in the matrix D. For example, since $x_1 - x_2 < 2$ and $x_2 - x_0 \le 2$, it must be the case $x_1 - x_0 < 4$. Since $x_0 = 0$, we see that $x_1 < 4$. Thus, we can change $\mathcal{D}_{1,0}$ to $(4, <)$ and obtain an alternative difference bound matrix for the same clock zone:

	0	1	2
0	$(0, \leq)$	$(-1, \leq)$	$(0, <)$
1	$(4, <)$	$(0, \leq)$	$(2, <)$
2	$(2, \leq)$	$(\infty, <)$	$(0, \leq)$

Clearly, the new matrix represents the same set of clock interpretations as the original matrix D.

In general, the sum of the upper bounds on the clock differences $x_i - x_j$ and $x_j - x_k$ is an upper bound on the clock difference $x_i - x_k$. This observation can be used to progressively tighten the difference bound matrix. If $x_i - x_j \prec_{i,j} d_{i,j}$ and $x_j - x_k \prec_{j,k} d_{j,k}$, then it is possible to conclude that $x_i - x_k \prec'_{i,k} d'_{i,k}$ where

$$d'_{i,k} = d_{i,j} + d_{j,k}$$

and

$$\prec'_{i,k} = \begin{cases} \leq & \text{if } \prec_{i,j} = \leq \text{ and } \prec_{j,k} = \leq \\ < & \text{otherwise} \end{cases}$$

Thus, if $(d'_{i,k}, \prec'_{i,k})$ is a tighter bound than $(d_{i,k}, \prec_{i,k})$, we should replace the latter by the former so that $\mathcal{D}_{i,k} = (d'_{i,k}, \prec'_{i,k})$. This operation is called *tightening* the difference bound matrix. We can repeatedly apply tightening to a difference bound matrix until further application of this operation does not change the matrix. The resulting matrix is a *canonical* representation for the clock zone under consideration. By following this procedure for the clock zone in the above example, we obtain the canonical difference bound matrix:

	0	1	2
0	$(0, \leq)$	$(-1, \leq)$	$(0, <)$
1	$(4, <)$	$(0, \leq)$	$(2, <)$
2	$(2, \leq)$	$(1, \leq)$	$(0, \leq)$

Note that a canonical difference bound matrix will satisfy the inequality $d_{i,k} \prec_{i,k} d_{i,j} + d_{j,k}$ for all possible values of the indices i, j, and k.

Finding the canonical form of a difference bound matrix can be automated by using the Floyd-Warshall algorithm [80], which has cubic complexity. The algorithm guarantees that all the possible combinations of indices are systematically checked to determine if further tightening is possible. We determine if a tighter bound can be obtained for $\mathcal{D}_{i,k}$ by checking if the inequality $d_{i,k} \prec_{i,k} d_{i,j} + d_{j,k}$ holds for all possible values of j. If the inequality does not hold for some value of j, then we replace $\mathcal{D}_{i,k}$ by $(d'_{i,k}, \prec'_{i,k})$ as described in the preceding paragraph.

After the difference bound matrix has been converted to canonical form, we can determine if the corresponding clock zone is nonempty by examining the entries on the main

diagonal of the matrix. If the clock zone described by the matrix is nonempty, all of the entries along the main diagonal will have the form $(0, \leq)$. If the clock zone is empty or unsatisfiable, there will be at least one negative entry on the main diagonal.

We describe now three operations on difference bound matrices [3, 99]. These operations correspond to the three operations defined on clock zones in the previous section.

- **Intersection.** We define $\mathcal{D} = \mathcal{D}^1 \wedge \mathcal{D}^2$. Let $\mathcal{D}^1_{i,j} = (c_1, \prec_1)$ and $\mathcal{D}^2_{i,j} = (c_2, \prec_2)$. Then $\mathcal{D}_{i,j} = (\min(c_1, c_2), \prec)$, where $\prec$ is defined as follows:
- If $c_1 < c_2$, then $\prec = \prec_1$.
- If $c_2 < c_1$, then $\prec = \prec_2$.
- If $c_1 = c_2$ and $\prec_1 = \prec_2$, then $\prec = \prec_1$.
- If $c_1 = c_2$ and $\prec_1 \neq \prec_2$, then $\prec = <$.
- **Clock reset.** Define $\mathcal{D}' = \mathcal{D}[\lambda := 0]$, where $\lambda \subseteq X$ as follows.
- If $x_i, x_j \in \lambda$ then $\mathcal{D}'_{i,j} = (0, \leq)$.
- If $x_i \in \lambda, x_j \notin \lambda$ then $\mathcal{D}'_{i,j} = \mathcal{D}_{0,j}$.
- If $x_j \in \lambda, x_i \notin \lambda$ then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,0}$.
- If $x_i, x_j \notin \lambda$, then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$.
- **Elapsing of time.** Define $\mathcal{D}' = \mathcal{D}^{\Uparrow}$ as follows:
- $\mathcal{D}'_{i,0} = (\infty, <)$ for any $i \neq 0$.
- $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$ if $i = 0$ or $j \neq 0$.

In each case the resulting matrix may fail to be in canonical form. Thus, as a final step we must reduce the matrix to canonical form. All three of the operations can be implemented efficiently. Moreover, the implementation of these operations is relatively straightforward to program.

We now see how the construction of the zone graph described in the previous section can be made more efficient. Clock zones are represented by difference bound matrices and the set $succ(\varphi, e)$ is computed by the three operations on difference bound matrices described above rather than by operations directly on clock zones. We will illustrate this procedure with the timed automaton in Figure 17.1. The initial state is given by $(s0,\ Z0)$, where $Z0$ is the clock zone $x = 0 \wedge y = 0$ which corresponds to the difference bound matrix:

	x_0	x	y
x_0	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$
x	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$
y	$(0, \leq)$	$(0, \leq)$	$(0, \leq)$

We follow the sequence of five steps given in Section 17.5. Only the normalized difference bound matrix obtained in each step is shown.

1. The invariant $I(s0)$ is $0 \leq x \wedge 0 \leq y \leq 5$, which is given by the matrix:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (\infty, <) & (0, \leq) & (\infty, <) \\ y & (5, \leq) & (5, \leq) & (0, \leq) \end{array}$$

We intersect D_0 with $I(s0)$ to obtain the zero matrix again.

2. Next, we let time elapse in the location $s0$ using the operator $\Uparrow$. The matrix for $(D_0 \wedge I(s_0))^{\Uparrow}$ is:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (\infty, <) & (0, \leq) & (0, \leq) \\ y & (\infty, <) & (0, \leq) & (0, \leq) \end{array}$$

3. We intersect with $I(s)$ again to find the set of clock assignments that still satisfy the invariant. The matrix for $(D_0 \wedge I(s_0))^{\Uparrow} \wedge I(s_0)$ is as follows:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (5, \leq) & (0, \leq) & (0, \leq) \\ y & (5, \leq) & (0, \leq) & (0, \leq) \end{array}$$

4. The guard g_a for the a transition from location $s0$ to location $s1$ is:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (-3, \leq) \\ x & (\infty, <) & (0, \leq) & (\infty, <) \\ y & (\infty, <) & (\infty, \leq) & (0, \leq) \end{array}$$

We intersect the current set of states with the guard g_a to obtain $((D_0 \wedge I(s_0))^{\Uparrow} \wedge I(s_0)) \wedge g_a$:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (-3, \leq) & (-3, \leq) \\ x & (5, \leq) & (0, \leq) & (0, <) \\ y & (5, \leq) & (0, \leq) & (0, \leq) \end{array}$$

5. Finally, we reset the clock y to get the matrix D_1:

$$\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (-3, \leq) & (0, \leq) \\ x & (5, \leq) & (0, \leq) & (5, <) \\ y & (0, \leq) & (-3, \leq) & (0, \leq) \end{array}$$

Note that the last difference bound matrix corresponds to the clock zone

$$Z1 \equiv 3 \leq x \leq 5 \ \wedge \ 3 \leq x - y \leq 5 \ \wedge \ y = 0.$$

Consequently, the successor state in the zone automaton is $(s1, \ Z1)$. Repeating the same sequence of steps, we obtain the remaining states of the zone automaton:

1. $(s0, \quad 4 \leq y \leq 5 \ \wedge \ 4 \leq y - x \leq 5 \ \wedge \ x = 0)$
2. $(s1, \quad 0 \leq x \leq 1 \ \wedge \ 0 \leq x - y \leq 1 \ \wedge \ y = 0)$
3. $(s0, \quad 5 \leq y \leq 8 \ \wedge \ 5 \leq y - x \leq 8 \ \wedge \ x = 0)$
4. $(s1, \quad x = 0 \ \wedge \ y = 0)$

The reachability computation terminates at this point because the state

$$(s1, \quad x = 0 \ \wedge \ y = 0)$$

is contained in

$$(s1, \quad 0 \leq x \leq 1 \ \wedge \ 0 \leq x - y \leq 1 \ \wedge \ y = 0).$$

Thus, no new states of $\mathcal{T}(A)$ will be obtained by computing successor states in the zone automaton.

17.7 Complexity Considerations

Because of Lemma 43, the complexity of checking reachability using the region graph construction is exponential in the number of clocks and also in the magnitude of the clocks. Similar complexity results can be obtained for temporal logic model checking. In practice, the region graph construction can be combined with symbolic model checking techniques based on BDDs. In this case the number of state variables can be quite large. However the number of clocks will usually be small (probably less than twenty).

Difference bound matrices are normally implemented using an explicit state representation. Consequently, the number of states that can be handled is much smaller in this approach. The number of clocks typically reflects the number of components operating

concurrently and will also be small. The advantage of this approach is that constraints are represented in terms of linear inequalities and, therefore, can be easily manipulated.

Various techniques discussed elsewhere in this book can be used to avoid the state explosion problem for continuous real-time systems. There has already been some research on combining the partial order reduction with these algorithms [222, 252, 253]. In addition, approximation techniques that can be used in conjunction with these algorithms have also been investigated [130]. Compositional reasoning should also be useful, but does not appear to have been investigated very thoroughly.

18 Conclusion

The techniques described in this book have already been used to find nontrivial errors in circuit and protocol designs. We have investigated the correctness of systems with realistic complexity like the Futurebus+ cache coherency protocol, described in Chapter 8.2. We believe that current model checking tools work sufficiently well to be of use in industry, and indeed, a number of companies such as AT&T, Fujitsu, Intel, IBM, Lucent, Motorola, and Siemens are beginning to use model checking as part of their design process. Because model checking avoids the construction of complicated proofs and provides a counterexample trace when some specification is not satisfied, we believe that circuit designers will find this technique relatively easy to learn and use. The tools are being adapted to use with industry-standard languages like VHDL and Verilog.

Nevertheless, there are a number of ways that model checkers can be improved to make them easier to use by engineers. Some of these improvements involve relatively straightforward extensions of current systems. For example, an obvious problem with current systems is how to make the specification language more expressive and easier to use. Some type of *timing diagram* notation may be more natural for engineers than CTL [227]. It may be possible either to translate timing diagrams systematically into temporal logic formulas or to check them directly using an algorithm similar to the one used by the model checker. A similar problem arises in finding a good way to display the counterexamples that are generated when a formula is not true. This feature is invaluable for actually finding the source of a subtle error in a circuit design. However, most current systems just print out a path in the state transition graph that shows how the error occurs. It is easy to imagine more perspicuous ways of displaying this information.

Other extensions require more theoretical work. An obvious direction for research is to develop even more concise techniques for representing boolean functions. The verification methods discussed in this book are not especially dependent upon the properties of binary decision diagrams. In fact, any representation of boolean functions that supports boolean operations and for which there are good simplification algorithms is a candidate for such a representation. As better representations are developed, they can easily be incorporated into model-checking algorithms.

The same encoding technique used in symbolic model checking can be used for other graph theoretic problems. For example, it is easy to write a formula in first-order μ-calculus that will be true when a graph is strongly connected. A model checking algorithm for this logic is given in [46]. An important question is how useful OBDDs and fixpoint techniques are for problems like finding minimum spanning trees, determining graph isomorphism, and so on.

In the long run, we expect to be able to handle more complex systems using abstraction and compositional reasoning techniques. Clearly, much additional research is needed on both of these topics. There have been some attempts to automate abstraction techniques

for hardware [181], though it is not clear how general these methods are. Some work has also recently appeared on automatic application of compositional methods [54]. There have been some recent successes in using abstract data domains for verifying processor designs [40, 88, 127], though these methods have not yet been combined smoothly with state-exploration techniques.

For verifying software, abstraction is even more critical, both to reduce the state space and to cope with infinite state spaces [149]. Analyzing software is becoming more important, especially with the growing interest in hardware/software codesign [156] for applications in consumer electronics products, and in complex safety-critical systems such as computer-controlled medical devices and air traffic control systems [241]. Newly developed software protocols for areas such as electronic commerce and authentication are also subject to very high reliability requirements [47, 70, 180, 188, 193, 194, 199]. Both hardware/software codesigns and computer security protocols are usually modeled as asynchronous systems. Partial order reduction techniques [126, 139, 209, 244] have been shown to be effective for verifying asynchronous systems such as communication protocols. We believe it is likely that such techniques will also prove effective for these new applications.

Much work is currently being devoted to methods for verifying real-time concurrent systems [134]. Such systems are particularly difficult to verify because their correctness depends on the actual times at which events occur. Verifiers based on a discrete-time semantics have already proved useful; for example, they have been used to check properties of the PCI bus [53]. It looks like it will be possible to extend such techniques to do performance analysis as well, for example, get maximum and minimum delays between two given events. Techniques based on continuous time have acquired a reputation of being computationally complex, but recent work suggests that these techniques may hold promise as well [10]. It is not yet clear which semantics will ultimately prove more useful, but probably both have their places. The two semantics can in fact be related, as shown by Burch [41], so it will probably be possible to think of the system in terms of whichever semantics seems most natural and then apply either a discrete or continuous time verification algorithm. Other possible directions for research involve using less precise but potentially faster algorithms. For example, when checking interface timing requirements that do not involve many complicated state dependencies, it is possible to use more efficient techniques [192]. Another idea is to try to combine static timing analysis algorithms [204] with state-exploration methods to try to find violations of timing constraints.

In Chapter 15, we showed how to apply induction to systems that could be described by context-free graph grammars. However, many important topologies, such as square meshes or arbitrary point-to-point networks, do not fall into this category. Thus, designs such as the Scalable Coherent Interface or SCI protocol (IEEE Standard 1596-1992) are currently not

amenable to inductive verification using model checking techniques. It might be possible to extend the network grammar approach to more powerful types of grammars.

Another direction for research is probablistic verification [14, 15, 85]. It may be possible to tell not only whether a failure is possible in a system, but also what the probability of the failure occurring is. If such an analysis were performed, failures with extremely low probabilities of actually occurring could be safely ignored. This type of analysis might also be used to estimate the mean time between failures for a system.

Currently, symbolic model checking is used to establish the correctness of a system and compute performance measures such as schedulability, response time, and system load. It may be feasible to extend current model checking tools to compute system dependability metrics such as Mean Time to Failure, Reliability, Availability, Minimum and Maximum Observed Time to Failure given the failure rates of the system components. The ultimate objective is to create a unified design environment that will enable users to obtain estimations of performance and dependability early in the design phase.

Finally, we believe that it is important to investigate how model checking techniques can be combined with theorem proving. We suspect that ultimately both model checkers and theorem provers will be needed to establish the correctness of complex circuits. Theorem provers seen necessary for reasoning about those parts of a complex microprocessor like the floating point arithmetic unit that require relatively deep mathematical knowledge. On the other hand, it seems unlikely that theorem proving systems will surpass model checking techniques for reasoning about complex hardware controllers in the near future. The problem is how to combine the two very different styles of reasoning into a single framework so that a user can smoothly integrate the results obtained by each. Some preliminary work in these directions has already begun [24, 100, 152, 163, 220].

References

[1] S. Aggarwal, R. P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C. H. West, eds., *Protocol Specification, Testing and Verification*, pp. 19–34. North Holland, 1983.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[3] R. Alur. Timed automata. NATO ASI Summer School on Verification of Digital and Hybrid Systems, 1998. (Available at www.cis.upenn.edu/alur/Nato97.ps.gz.)

[4] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qudeer, and S. K. Rajamani. Partial–order reduction in symbolic state space explosion. In O. Grumberg, ed., *9th International Conference on Computer Aided Verification*, *LNCS* 1254, pp. 340–351, Springer, 1997.

[5] R. Alur and D. L. Dill. Automata-theoretic verification of real-time systems. In Constance Heitmeyer and Dino Mandrioli, eds., *Formal Methods for Real-Time Computing*, pp. 55–80. Wiley, 1996.

[6] R. Alur and T. A. Henzinger, eds. *Proceedings of the 1996 Workshop on Computer-Aided Verification*, *LNCS* 1102. Springer, 1996.

[7] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[8] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pp. 414–425. IEEE Computer Society Press, 1990.

[9] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science* 126(2): 183–235.

[10] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In R. Alur, T. A. Henzinger, and E. D. Sontag, eds., *Hybrid Systems III: Verification and Control*, *LNCS* 1066, pp. 220–231. Springer, 1995

[11] H. R. Andersen. Model checking and boolean graphs. In B. Krieg-Bruckner, ed., *Proceedings of the Fourth European Symposium on Programming*, *LNCS* 582, pp. 1–19. Springer, 1992.

[12] K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *IPL* 15: 307–309.

[13] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Equivalences for fair Kripke structures. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, *LNCS* 820, pp. 364–375. Springer, 1994.

[14] A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In Wolper [249], pp. 155–166.

[15] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, eds., *24th International Colloquium on Automata, Languages, and Programming (ICALP '97)*, *LNCS*, 1256, pp. 430–440. Springer, 1997.

[16] F. Balarin and A. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In Dill [97], pp. 235–246.

[17] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Courcoubetis [83], pp. 29–40.

[18] D. L. Beatty, R. E. Bryant, and C-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1991.

[19] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica* 20(1983): 207–226.

[20] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In von Bochmann and Probst [243], pp. 260–273.

[21] W. Bernard and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, pp. 294–303. IEEE Computer Society, 1996.

[22] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata theoretic approach to branching time model checking. In Dill [97], pp. 142–155.

[23] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE International Conference on Computer Design*, 1990.

[24] N. Bjorner, et al. Step: The Stanford temporal prover—user's manual. Technical Report STAN-CS-TR-95-1562, Department of Computer Science, Stanford University, November 1995.

[25] G. von Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers* C-31(3).

[26] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, ed., *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[27] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* IEEE Computer Society Press, 1990.

[28] M. C. Browne and E. M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, pp. 269–292. IFIP, 1987.

[29] M. C. Browne, E. M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proceedings of the 1985 International Conference on Computer Design*, pp. 545–548. IEEE, 1985..

[30] M. C. Browne, E. M. Clarke, and D. L. Dill. Automatic circuit verification using temporal logic: Two new examples. In G. J. Milne and P. A. Subrahmanyam, eds., *Formal Aspects of VLSI Design*. Elsevier, 1986.

[31] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers* C-35(12): 1035–1044.

[32] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59(1–2): 115–131.

[33] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation* 81(1): 13–31.

[34] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35(8): 677–691.

[35] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers* 40(2): 205–213.

[36] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24(3): 293–318.

[37] R. E. Bryant and C-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In Clarke and Kurshan [71], pp. 33–43.

[38] J. A. Brzozowski and C. J. H. Seger. Advances in asynchronous circuit theory. Part II: Bounded inertial delay models, MOS circuits, design techniques. *Bulletin of the European Association for Theoretical Computer Science* 43(3): 199–263.

[39] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pp. 1–11. Stanford University Press, 1960.

[40] J. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In Dill [97], pp. 68–81.

[41] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[42] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 403–407. IEEE, 1991.

[43] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, eds., *Proceedings of the 1991 International Conference on VLSI*, pp. 49–58. August 1991.

[44] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits* 13(4): 401–424.

[45] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 46–51. IEEE, 1990.

[46] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2): 142–170.

[47] M. Burrow, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Transactions on Computer Systems* 8(1): 18–36.

[48] R. M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress 74*, pp. 308–312. North Holland, 1974.

[49] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time System*. PhD thesis, Carnegie Mellon University, 1996.

[50] S. V. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In *First AMAST International Workshop in Real-Time Systems*. Springer, 1993.

[51] S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *ACM Workshop on Languages Compilers and Tools for Real-Time Systems*, 1995.

[52] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*. IEEE, 1994.

[53] S. V. Campos, E. M. Clarke, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proceedings of the IEEE International Conference on Computer Design*, pp. 73–79. IEEE, 1995.

[54] M. Chiodo, T. R. Shiple, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Automatic compositional minimization in CTL model checking. In ICCAD92 [146], pp. 172–178.

[55] C-T. Chou and D. Peled. Verifying a model-checking algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 241–257. Springer, 1996.

[56] L. Claesen, ed. *Proceedings of the 11th International Symposium on Computing Hardware Description Languages and their Applications*. North Holland, 1993.

[57] D. Clarke, H. Ben-Abdallah, I. Lee, H. Xie, and O. Sokolsky. XVERSA: an integrated graphical and textual toolset for the specification and analysis of resource-bound real-time systems. In *Computer-Aided Verification*, pp. 402–405. Springer, 1996.

[58] E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design* 9: 77–104.

[59] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, *LNCS* 354, pp. 428–437. Springer, 1988.

[60] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, eds., *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, *LNCS* 407, pp. 103–116. Springer, 1990.

[61] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, *LNCS* 131. Springer, 1981.

[62] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Language*, January 1983.

[63] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8(2): 244–263.

[64] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [83], pp. 450–462.

[65] E. M. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* 10(1): 47–71.

[66] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In Claesen [56].

[67] E. M. Clarke, O. Grumberg, and S. Jha. Parametrized networks. In S. Smolka and I. Lee, eds., *Proceedings of the 6th International Conference on Concurrency Theory*, *LNCS* 962, pp. 395–407. Springer, 1995.

[68] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(5): 726–750.

[69] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems* 16(5): 1512–1542.

[70] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[71] E. M. Clarke and R. P. Kurshan, eds. *Workshop on Computer-Aided Verification. 2nd International Conference, CAV'90. Proceedings*, *LNCS* 531. Springer, 1990.

[72] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In J. A. Darringer and F. J. Rammig, eds., *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and Their Applications*, pp. 281–295. North Holland, 1989.

[73] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In v. Bochmann and Probst [243], pp. 410–422.

[74] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design* 2(2): 121–147.

[75] R. W. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica* 27: 725–747.

[76] R. W. Cleaveland, P. Lewis, S. Smolka, and O. Sokolsky. The concurrency factory: a development environment for concurrent systems. In Alur and Henzinger [6], pp. 398–401.

[77] R. W. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In Sifakis [231], pp. 24–37.

[78] R. W. Cleaveland and S. Sims. The NCSU concurrency workbench. In Alur and Henzinger [6], pp. 394–397.

[79] P. Clements, C. Heitmeyer, G. Labaw, and A. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

[80] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1989.

[81] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Sifakis [231], pp. 365–373.

[82] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Clarke and Kurshan [71], pp. 23–32.

[83] C. Courcoubetis, ed. *Proceedings of the 5th Workshop on Computer-Aided Verification*, June/July 1993.

[84] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1: 275–288.

[85] C. Courcoubetis and M. Yannakakis. The complexity of probablistic verification. *Journal of the ACM* 42(4): 857–907.

[86] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedimgs of the 4th Annual ACM Symposium on Principles of Programming Language*, pp. 238–252, January 1977.

[87] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Language*, pp. 269–282, January 1979.

[88] D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In Dill [97], pp. 247–259.

[89] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technical University of Eindhoven, Eindhoven, 1995.

[90] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In *5th Conference on Computer-Aided Verification*, *LNCS* 697, pp. 479–490. Springer, 1993.

[91] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(2): 253–291.

[92] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, *LNCS* 1066, pp. 208–219. Springer, 1996.

[93] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the* 16th *Real-Time Systems Symposium*, pp. 66–75. IEEE Computer Society Press, 1995.

[94] J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds. *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, *LNCS* 430. Springer,1989.

[95] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8): 453–457.

[96] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[97] D. L. Dill, ed. *Proceedings of the 1994 Workshop on Computer-Aided Verification*, *LNCS* 818. Springer, 1994.

[98] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings* Part E 133(5), 1986.

[99] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, ed., *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, *LNCS* 407, pp. 197–212. Springer, 1989.

[100] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Wolper [249], pp. 54–69.

[101] P. Dixon. Multilevel cache architectures. Minutes of the Futurebus+ Working Group Meeting, December 1988.

[102] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* 3: 245–260.

[103] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[104] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *LNCS* 85, *Automata, Languages and Programming*, pp. 169–181. Springer, 1980.

[105] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM* 33: 151–178.

[106] E. A. Emerson and C-L. Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, pp. 84–96. ACM Press, January 1985.

[107] E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In LICS86 [174], pp. 267–278.

[108] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasen. Quantitative temporal reasoning. In Clarke and Kurshan [71], pp. 136–145.

[109] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pp. 85–94. ACM Press, 1995.

[110] E. A. Emerson and K. S. Namjoshi. Automated verification of parameterized synchronous systems. In Alur and Henzinger [6] pp. 87–98.

[111] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In Courcoubetis [83], pp. 463–478.

[112] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model-checking for fragments of mu-calculus. In Courcoubetis [83] pp. 385–396.

[113] J. C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Alur and Henzinger [6], pp. 348–359.

[114] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, ed., *Mathematical Aspects of Computer Science*, pp. 19–32. American Mathematical Society, 1967.

[115] N. Francez. *The Analysis of Cyclic Programs*. PhD thesis, The Weizmann Institute of Science, 1976.

[116] N. Francez. *Fairness*. Springer, 1986.

[117] A. N. Fredette and R. W. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[118] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, 1988.

[119] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *Proceedings of the IFIP WG10.2 International Conference on Hardware Description Languages and their Applications*. 1985.

[120] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pp. 163–173. ACM, 1980.

[121] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[122] R. Gerber and I. Lee. A proof system for communicating shared resources. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1990.

[123] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM* 39: 675–735.

[124] R. Gerth, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pp. 3–18. Chapman & Hall, 1995.

[125] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd Workshop on Computer-Aided Verification*, *LNCS* 531, pp. 176–185. Springer, 1990.

[126] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Conference on Computer-Aided Verification*, *LNCS*, 697, pp. 438–449. Springer, 1993.

[127] S. Graf. Verification of a distributed cache memory by using abstractions. In Dill [97], pp. 207–219.

[128] S. Graf and B. Steffen. Compositional minimization of finite state processes. In Clarke and Kurshan [71], pp. 186–196.

[129] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* 16: 843–872.

[130] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design* 11(2): 157–185.

[131] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering* 20(1): 13–28.

[132] R. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [6], pp. 423–427.

[133] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal* 69(1): 45–59.

[134] C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*. Wiley, 1996.

[135] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation* 3(2): 193–244.

[136] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the Association for Computing Machinery* 12(10): 322–329.

[137] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[138] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[139] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques 1994*, pp. 197–211. Chapman & Hall, 1994.

[140] G. J. Holzmann and D. Peled. The state of spin. In *CAV'96: 8th International Conference on Computer Aided Verification*, *LNCS* 1102, pp. 385–389. Springer, 1996.

[141] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Second SPIN Workshop*, pp. 23–32. AMS, 1996.

[142] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[143] P. Huber, A. Jensen, L. Jepsen, and K. Jensen. Towards reachability trees for high-level Petri nets. In G. Rozenberg, ed., *Advances on Petri Nets*, pp. 215–233. 1984.

[145] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, 1977.

[146] *IEEE Computer Society. 1992 Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, 1992.

[147] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*. IEEE Computer Socity Press, 1992. IEEE Standard 896.1–1991.

[148] C. W. Ip and D. L. Dill. Better verification through symmetry. In Claesen [56].

[149] D. Jackson. Abstract model checking of infinite specifications. In *Proceedings of Formal Methods Europe*, Barcelona, Oct. 1994.

[150] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pp. 321–332. North-Holland, 1983.

[151] B. Josko. Verifying the correctness of AADL-modules using model checking. In de Bakker et al. [94].

[152] J. J. Joyce and C-J. H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. Association for Computing Machinery, 1993.

[153] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, *LNCS* 354, pp. 489–507. Springer, 1988.

[154] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[155] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional laguages. In O. Grumberg, ed., *9th International Conference on Computer Aided Verification (CAV'97)*, *LNCS* 1254, pp. 424–435. Springer, 1997.

[156] K. Keutzer. Hardware-software co-design and ESDA. In *Proceedings of the 31th Design Automation Conference*, pp. 435–436. June 1994.

[157] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science* 27: 333–354.

[158] F. Kröger. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8(3): .

[159] O. Kupferman and M. Y. Vardi. Verification of fair transition systems. In Alur and Henzinger [6], pp. 372–382.

[160] R. P. Kurshan. Analysis of discrete event coordination. In de Bakker et al. [94], pp. 414–453.

[162] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, pp. 170–172. Princeton University Press, 1994.

[163] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Courcoubetis [83], pp. 166–180.

[164] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 1384, pp. 345–357. Springer, 1998.

[165] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pp. 239–247. ACM, 1989.

[166] L. Lamport. "Sometimes" is sometimes "Not Never." In *Annual ACM Symposium on Principles of Programming Language*, pp. 174–185. ACM, 1980.

[167] L. Lamport. What good is temporal logic. In *Information Processing 83*, pp. 657–668. Elsevier, 1983.

[168] K. G. Larsen. Modal specifications. In Sifakis [231], pp. 232–246.

[169] K. G. Larsen. Efficient local correctness checking. In v. Bochmann and Probst [243], pp. 30–43.

[170] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th Real-Time Systems Symposium*, pp. 76–87. IEEE Computer Society Press, 1995.

[171] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1990.

[172] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In *Foundations of Real-Time Computing—Scheduling and Resource Management*. Kluwer Academic, 1991.

[173] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Language*, pp. 97–107. ACM, 1985.

[174] *Proceedings of the 1st Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986.

[175] B. Lin and A. R. Newton. Efficient symbolic manipulation of equvialence relations and classes. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, pp. 46–61. January 1991.

[176] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20(1): 46–61.

[177] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in Ada: a case study. In *IEEE Real-Time Systems Symposium*. IEEE Computer Socity Press, 1991.

[178] D. Long, A. Browne, E. Clarke, S. Jha, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In Dill [97], pp. 338–350.

[179] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.

[180] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 1055, pp. 147–166. Springer, 1996.

[181] E. Macii, B. Plessier, and F. Somenzi. Verification of systems containing counters. In ICCAD92 [146], pp. 179–182.

[182] S. MacLane and G. Birkhoff. *Algebra*. MacMillan, 1968.

[183] A. Mader. Tableau recycling. In v. Bochmann and Probst [243], pp. 330–342.

[184] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, eds., *VLSI Systems and Computations*. Computer Science Press, 1981.

[185] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincenteli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, pp. 6–9. 1988.

[186] Z. Manna and A. Pnueli. *Temporal Verifications of Reactive Systems—Safety*. Springer, 1995.

[187] R. Marelly and O. Grumberg. GORMEL—Grammar ORiented ModEL checker. Technical Report 697, The Technion, October 1991.

[188] W. Marrero, E. M. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[189] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In H. Fuchs, ed., *Proceedings of the 1985 Chapel Hill Conference on VLSI*, 1985.

[190] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In v. Bochmann and Probst [243], pp. 164–177.

[191] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[192] K. L. McMillan and D.L. Dill. Algorithms for interface timing verification. In ICCAD92 [146], pp. 48–51.

[193] C. Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the 1994 Computer Security Foundations Workshop*. IEEE Computer Society Press, 1994.

[194] J. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pp. 251–260. IEEE Computer Society Press, 1995.

[195] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pp. 481–489. 1971.

[196] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[197] B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science* 38: 269–291.

[198] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering* SE-7, No. 4: 417–426.

[199] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1997.

[200] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton University Press, 1956.

[201] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[202] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[203] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica* 18: 265–287.

[204] J. K. Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on Computer-Aided Design* 4(3): 336–349.

[205] W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California at Los Angeles, 1981.

[206] R. Paige and R. E. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing* 16(6): 973–989.

[207] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pp. 167–183. Springer, 1981.

[208] D. Peled. All from one, one for all: on model checking using representatives. In Courcoubetis [83], pp. 409–423.

[209] D. Peled. Combining partial order reductions with on-the-fly model-checking. In Dill [97], pp. 377–390.

[210] D. Peled. Verification for robust specification. In Elsa Gunter, ed., *Conference on Theorem Proving in Higher Order Logic*, pp. 231–241. Springer, 1997.

[211] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the nexttime operator. *Information Processing Letters*, 1997.

[212] D. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *CONCUR'96, 7th International Conference on Concurrency Theory, LNCS* 1119, pp. 596–610. Springer, 1996.

[213] C. Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In Clarke and Kurshan [71], pp. 54–64.

[214] C. Pixley, G. Beihl, and E. Pacas-Skewes. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of the International Conference on Computer Design*, pp. 245–249, Cambridge, MA, October 1991.

[215] C. Pixley, S-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*, pp. 620–623, June 1992.

[216] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pp. 46–57. IEEE Computer Society Press, 1977.

[217] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science* 13: 45–60.

[218] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, ed., *Logics and Models of Concurrent Systems, NATO ASI* 13. Springer, 1984.

[219] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pp. 337–350.

[220] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In Wolper [249], pp. 84–97.

[221] R. Rajkumar. *Task Synchronization in Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1989.

[222] Tomas G. Rokicki and Chris J. Myers. Automatic verification of timed circuits. In Dill [97], pp. 468–480.

[223] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, ed., *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pp. 353–378. Prentice Hall, 1994.

[224] V. Roy and R. de Simone. Auto/Autograph. In Clarke and Kurshan [71], pp. 235–250.

[225] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer Aided Design*, Santa Clara, Ca., November 1993.

[226] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pp. 319–327. IEEE Computer Society, 1988.

[227] R. Schlor and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *EDAC 93*, 1993.

[228] L. Sha, M. H. Klein, and J. B. Goodenough. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing—Scheduling and Resource Management*. Kluwer Academic, 1991.

[229] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In Sifakis [231], pp. 157–165.

[230] G. Shurek and O. Grumberg. The modular framework of computer-aided verification: Motivation, solutions and evaluation criteria. In Clarke and Kurshan [71], pp. 214–223.

[231] J. Sifakis, ed. *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, LNCS* 407. Springer, 1989.

[232] A. P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.

[233] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM* 32(3): 733–749.

[234] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science* 49: 217–237.

[235] Richard H. Sloan and Ugo Buy. Stubborn sets for real-time Petri nets. *Formal Methods in System Design* 11(1): 23–40.

[236] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science* 89(1): 161–177.

[237] I. Suzuki. Proving properties of a ring of finite-state machines. *IPL* 28: 213–214.

[238] N. Suzuki, ed. *Symbolic Computation Algorithms on Shared Memory Multiprocessors*. MIT Press, 1992.

[239] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing* 1: 146–160.

[240] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5: 285–309.

[241] (TCAS) Minimum operational performance standards for traffic alert and collision avoidance system (TCAS) airborne equipment, volume II. Radio Technical Commission for Aeronautics, September 1990.

[242] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.

[243] G. von Bochmann and D. K. Probst, eds. *Workshop on Computer-Aided Verification. Fourth International Workshop, CAV'92. Proceedings*, *LNCS* 663. Springer, 1992.

[244] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, *LNCS* 531, pp. 156–165. Springer 1990.

[245] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In LICS86 [174], pp. 332–344.

[246] P. D. Vigna and C. Ghezzi. Context-free graph grammars. *Information and Computation* 37: 207–233.

[247] G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the 16th International Colloquium on Automata, Languages, and Programming*, *LNCS* 372. pp. 761–772. Springer, 1989.

[248] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Language*, January 1986.

[249] P. Wolper, ed. *Proceedings of the 1995 Workshop on Computer-Aided Verification*, *LNCS* 939. Springer, 1995.

[250] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In Sifakis [231], pp. 68–80.

[251] J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[252] T. Yoneda and B. Schlingloff. Efficient verification of parallel real-time systems. *Formal Methods in System Design* 11(2): 197–215.

[253] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In Courcoubetis [83], pp. 321–332.

Index

μ-calculus, 69, 97–108
  complexity, 101, 103, 108
  CTL translated into, 107–108
  OBDDs, 104–107

Abstractions, 193–213
  approximations, computing, 199–202
  approximations, exact, 202–203
  automation, 293–294
  data abstraction, 195–213
  examples, 204–213
  future research on, 293–294
  infinite families, 242–248
  introduction to, 10
  symbolic, 210–213
  usefulness of, 195–196
Accepting, in automata theory, 121–122
Action transitions, 268
ACTL, 31, 32
  abstractions, 196
  bisimulation, 178
  compositional reasoning, 186–191
  tableau construction, 180–184, 186
ACTL*, 31, 32
  abstractions, 197, 202
  simulation, 177, 232
Aircraft controller example, 259–264
Alternation depth, 97, 100, 108
ALU example, 75–77, 85–87
Always operator, 27
Ample sets, 9, 143–154
  calculation of, 144, 154–160
  selection of, 147–151, 154–155
Apply algorithm, 56–57
Assignment statement, 21
Assume-guarantee reasoning, 10, 181, 185–186, 189–190. *See also* Compositional reasoning
Asynchronous execution, 17
Asynchronous systems, 8, 19–20
Atom, 43
Atomic transitions, 16
Automata theory, 121–140
  advantages of, 123
  deterministic automata, 127–128
  emptiness checking, 129–132
  finite and infinite words, 121–122
  generalized automata, 128–129
  infinite families, 245–248
  intersection and complementation, 124–127
  language containment checking, 139–140
  LTL, 125, 132–138
  nondeterministic automata, 127–128
  OBDDs, 57
  on-the-fly model checking, 138–139
  partial order reduction, 167–168
  timed automata, 265–291
Automatic methods, advantages of, xi-xii, xiii, 3
Automorphisms, 216–221

Binary decision diagrams (BDDs), 52–59
  isomorphic, 53
  ordered (*See* Ordered binary decision diagrams)
  partitioned transition relations, 84–85
  transformation rules, 53
  usefulness of, xiv
Binary decision trees, 51–52, 240, 242
Bisimulation, 171–180
  abstractions, 194–195, 202–203
  largest, 179–180
  quotient models, 220
Bitwise operations, and abstraction, 208–210
Blocks, 146, 226
Boolean attributes, 116
Boolean formulas, 7. *See also* Ordered binary decision diagrams
Bounded until operator, 255–256
Bound variables, 98
Branching structure, 5, 6, 30
Breadth first search, 143, 155
Büchi automata, 122, 123
  emptiness checking, 129–132
  generalized, 128–129
  intersection and complementation, 124–127
  language containment, 139–140
  LTL, 132–138
  nondeterministic, 127–128
Bus architecture, 112–120. *See also* Futurebus+ standard
Busy waiting statement, 24

Cache coherence protocol, 7–8
  invariants, 235–238
  SMV, 112–120, 235
  symmetry, 228–230
Canonical representations, 52–53, 288–289
Chinese remainder theorem, 206
Clock constraints, 266–267
Clock regions, 274–280
Clock reset operation, 281, 289
Clocks, 266–291
Clock zones, 280–291
Closed formulas, 98
Code regions, 109–110
Commutativity, 141, 144–145, 147
Complementation, 124–127
Completeness, 4, 5, 6, 41, 180

Compositional reasoning, 185–191
  automation, 186–187, 294
  continuous real time, 292
  example, 190–191
  future research on, 293
  justification, 189–190
Computability, theory of, 3
Computation, as sequence of states, 13
Computation Tree Logic. *See* CTL
Computation trees, 27
Concurrent programs, 22–24
Concurrent systems
  first order representations, 14–16
  modes of execution, 17
  types of, 17–24
Conditional statement, 22
Condition counting algorithms, 259
Cone of influence reduction, 193–195
Congruences, and abstraction, 203, 205–208
Conjunctive partitioning, 79, 80–83
Context-free grammars, 238, 248, 294. *See also* Network grammars
Continuous real time, 265–292
  clock regions, 274–280
  clock zones, 280–291
  complexity, 291–292
  difference bound matrices, 283, 287–291
  parallel composition, 268–269
  region graph, 279–280, 291
  timed automata, 265–291
Cost metrics, 84
Counterexamples
  advantages of, 4
  automata theory, 125, 129, 131, 139
  display of, 293
  partial order reduction, 144
  simulation, 177
  SMV, 112
  symbolic model checking, 71–74
CPU controller verification example, 190–191
Critical region, 25, 109, 111
CTL (Computation Tree Logic), 5, 7, 35–41
  μ-calculus, 107–108
  bisimulation, 175, 177
  fairness, 32–33
  operators, 30–31
  RTCTL extension, 255–256
  symbolic model checking, 66–87
CTL* (Computation Tree Logic*), 6, 27–30
  abstractions, 195, 202
  algorithm, 46–49
  bisimulation, 171–175, 195, 202
  fairness, 32–33
  indexed, 231
  infinite families, 231–232
  operators, 27–28
  simulation preorder, 232
  sublogics, 30–32
  symmetry, 221
  syntax and semantics, 28–30
Cycle condition, 150–151, 155, 159–160
Cycles, in group theory, 216

Data abstraction, 195–213
  approximations, computing, 199–202
  approximations, exact, 202–203
  examples, 204–213
  pushing inward, 200–202
Deductive verification, 2–3. *See also* Proof systems
Delay transitions, 267
Dependability metrics, 295
Dependency relation, 141, 144–145, 157
Depth first search (DFS), 129–132, 139
  partial order reduction, 143, 147–154, 160, 168
Design validation, problem of. *See* Model checking
Deterministic automata, 127–128
Deterministic structures, 179, 180
Deterministic transitions, 109, 141
DFS. *See* Depth first search
Difference bound matrices, 283, 287–291
Digital circuits, 17–20
Directed acyclic graph, 52
Disabled transitions, 141
Discrete real-time systems, 254, 255–264, 265
Disjunctive partitioning, 79, 80
Domain of interpretation, 14
Dynamic reordering, 54–55

Efficiency
  abstractions, 197
  automata theory, 138–139
  partial order reduction, 143
  symbolic model checking, 80–87
Elapsing of time operation, 281–282, 289
Embedded systems, 2
Emptiness checking, of Büchi automata, 129–132
Enabledness, 144
Enabled transitions, 141, 143–144, 147–151. *See also* Ample sets
Entry point, of a statement, 20
Equivalences
  bisimulation (*See* Bisimulation)
  language, 179–180, 232
  simulation (*See* Simulation preorder)
  stuttering, 146–154, 163–164
Error trace, 4. *See also* Counterexamples

Eventuality sequence, 43
Eventually operator, 27
Exhaustive exploration, xi
Exit point, of a statement, 20
Explosion. *See* State explosion

Failure, probability of, 295
Fair bisimulation relation, 175
Fairness, 32–33
Fairness constraints, 20, 32–33, 40–41, 59
  bisimulation, 171, 175, 179, 180
  compositional reasoning, 186, 187
  simulation, 178, 186
  SMV, 111
  symbolic model checking, 68–71, 72–74, 91–95
Fair semantics, 32
Fair simulation relation, 178, 186
Finite automaton, 121. *See also* Automata theory
Finite versus infinite state systems, 3
Finite words, 121–122. *See also* Automata theory
Fixpoints, 61–66
  μ-calculus, 97–108
  fairness constraints, 69–70
  future research on, 293
  introduction to, 7
Flagged node, 130
Floyd-Warshall algorithm, 288
Formal verification, need for, 1–2. *See also* Model checking
Free variables, 98
Full symmetric groups, 215, 226, 228
Fully expanded state, 147
Futurebus+ standard, 7–8
  invariants, 235–238
  SMV, 112–120, 235
  symmetry, 228–230

Globally operator, 27
Grammars
  future research on, 294–295
  infinite families and, 238–248
Granularity, 16–17
Graph grammars, 238–239
Graph isomorphism problem, 224–225
Graphs
  decomposition into SCCs, 36, 44
  encoding techniques, 293
  region, 279–280, 291
  tableau construction, 42, 87–95
  zone, 283–284, 289
Greedy algorithm, 83, 84
Group theory, 215–221
Guard, 165, 266

Hashed node, 130

ICTL* (indexed CTL*), 231
Independence relation, 144–145
Independent events, 9
Induction technique, 11
Infinite families, 231–252
  example, Futurebus+, 235–238
  example, token ring, 232–233, 248–252
  grammars, graph and network, 238–248
  invariants, 232–238, 241–242
  temporal logic, 231–232
  undecidability, 231, 248–252
Infinite versus finite state systems, 3
Infinite words, 121–122. *See also* Automata theory
Initial states, 14–15, 27, 35
Instantaneous transitions, 265
Interleaved composition, 109
Interleaved execution, 17
Interleaving model, 9, 19, 141, 142
Internet, 2
Intersection operation
  automata theory, 124–127
  clock zones, 281
  difference bound matrices, 289
Invariance groups, 219–221
Invariant processes, 11
Invariant rule, 233, 241
Invariants
  clock constraints, 266
  infinite families, 232–238, 241–242
Invisibility, 145, 147, 149–150

Kripke structures, 13
  μ-calculus modification, 98
  abstractions, 196–197, 199–200, 204
  automata theory, 123
  automorphism group, 216–221
  CTL* semantics, 29
  definition, 14
  example, 26
  extraction, 14–17
  fairness, 33
  infinite families, 231–232, 238–242
  model checking algorithms, 35–49
  OBDDs and encoding, 51, 57–59
  partial order reduction, 141
  symbolic model checking, 61–95

Labeling transformations, 20–23
Language
  automata theory, 121–122, 139–140
  equivalence, 179–180, 232
  synchronous, 203–204

Leader election algorithm, 165–166
Length, of a path, 142
Lichtenstein-Pnueli algorithm, 42, 87
Linear structure, 5, 6, 30
Linear Temporal Logic. *See* LTL
Locations, and timed automata, 265–291
Lock statement, 24
Logarithms, 208
LTL (Linear Temporal Logic), 5, 6, 30
  automata theory, 125, 132–138
  invariants, 232
  model checking algorithm, 41–46
  partial order reduction, 146–154, 159, 164
  symbolic model checking, 87–95

Maximum delay algorithm, 257–259
Mean time between failures, 295
Message exchange, 17
Microwave oven example, 38–39, 47–49
Minimum delay algorithm, 257
Model checking
  advantages of, 1, 3, 293
  compared to other methods, 2–3
  future research on, 293–295
  limitations of, xiii, 1, 293
  need for, xi, 1–2
  on-the-fly, 138–139
  process of, 4
  using representatives, 141 (*See also* Partial order reduction)
Model checking problem, 35
Modeling, as step in model checking, 4, 13–26
  example, 24–26
  first-order representations, 14–16
  granularity, 16–17
  Kripke structure defined, 14
  types of concurrent systems, 17–24
Modular structure, 10
Modules, in SMV, 109
Monitor task servers, 260
Monotonicity
  of composition, 233
  syntactic, 98
Moore machine, 204
Mutual exclusion, 24
  SMV, 109–112

Negation normal form, 132
Network grammars, 239–248, 295. *See also* Context-free grammars
Never claim construct, 167
Next state variables, 15
Next time operator, 27
Nodes, 130, 133
Noncritical region, 25, 109, 111
Nondeterministic Büchi automata, 127–128
Nondeterministic structures, 179, 180
Nondeterministic transitions, 109
Nonterminal vertices, 51–52

OBDDs. *See* Ordered binary decision diagrams
On-the-fly model checking, 138–139
  partial order reduction, 143, 159, 164, 168
Orbit problem, 224–225
Orbit relation, 222–223, 226–228
Ordered binary decision diagrams (OBDDs), 51–59
  μ-calculus, 104–107
  abstractions, 196, 203–204, 210–211
  algorithms, 66–95
  bisimulation, 180
  boolean representation, 51, 53–57
  future research on, 293
  introduction to, 7–8
  Kripke structures, encoding of, 51, 57–59
  logical operations, 55–57
  ordering selection, heuristics for, 54–55
  ordering selection, partitioning and, 82–85
  partitioned transition relations, 79–85
  size of, 53–54, 84, 226–228
  symmetry, 222–223, 226–228

Parallel composition, 187, 188–189, 268–269
Parameterized designs, 231. *See also* Infinite families
Partial order reduction, 141–170
  ample set calculation, 144, 154–160
  ample set selection, 147–151, 154–155
  asynchronous system concurrency, 142–144
  commutativity, 144–145, 147
  correctness, 160–164
  dependency relation, 141, 144–145, 157
  future research on, 294
  independence relation, 144–145
  introduction to, 8–9
  invisibility, 145, 147, 149–150
  LTL, 146–154, 159, 164
  SPIN system, 159, 164–170
  stuttering equivalence, 146–154, 163–164
  usefulness of, xiv, 141
Partitioned transition relations, 79–87
Passable guard, 165
Path formulas, 28–29
Path quantifiers, 27
Permutation groups, 215–216
Persistent sets, 9
Point-to-point networks, arbitrary, 294
Positive normal form, 31
Predicate transformers, 7, 61–66

Preorder, simulation, 176–180, 188–189, 232–234
Present state variables, 15
Preservation, strong and weak, 232
Priority inversion, 259, 262–264
Probablistic verification, 295
Process, 22, 24
Process synchronization, 24
Program counter, 21, 156
Programs
  concurrent, 22–24
  sequential, 20–22
  translation example, 24–26
PROMELA language, 164–168
Proof systems, xiii, 3, 295. *See also* Deductive verification
PSPACE-completeness, 6, 41, 180
Pushing inward, and data abstraction, 200–202

Quantified boolean formulas, 66–67
Quantitative temporal analysis, 256–264
Quotient models, 218–221, 222–223
Quotient structure, 218–219

Rate-monotonic scheduling (RMS), 253–254, 261
Reachability, 154, 268–292
Reactive systems, 13, 27
Real-time systems, 253–292
  clock regions, 274–280
  clock zones, 280–291
  complexity, 291–292
  continuous, 265–292
  difference bound matrices, 283, 287–291
  discrete, 254, 255–264, 265
  examples, 259–264, 269–273
  future research on, 294
  limitations on checking of, 254
  parallel composition, 268–269
  quantitative temporal analysis, 256–264
  region graph, 279–280, 291
  RMS theory, 253–254, 261
  RTCTL model checking, 255–256
  timed automata, 265–291
Reduced state graph, 143. *See also* Partial order reduction
Region graph, 279–280, 291
Relational product, 77–87
Release operator, 28
Representatives, 141. *See also* Partial order reduction
Restricted path formula, 41, 87
Result cache, 56
RMS (rate-monotonic scheduling), 253–254, 261
Rotation groups, 226–228
RTCTL model checking, 255–256

SCCs (strongly connected components), 36, 44, 129
Schedulability, 253, 254, 255, 262–264. *See also* Real-time systems
Security protocols, 3
Self-fulfilling SCC, 44
Sequential composition statement, 22
Sequential programs, 20–22
Shannon expansion, 55–56
Shared variables, 17, 24
Simulation, method of, xi, xiii, 2
Simulation preorder, 176–180, 188–189, 232–234
Simulation relation, 176
Single bit abstraction, 208–210
Skip statement, 21
Sleep sets, 9
SMV (Symbolic Model Verifier), 7, 8, 109–120
  example, 112–120
  features, 109–112
Snarfing, 114
Solvability. *See* Undecidability
Specification, as step in model checking, 4
SPIN system, 159, 164–170
Square meshes, 294
State explosion, 9, 10–11
  challenge of, xiii, 1, 8
  concurrent events, all orderings of, 142
  continuous real time, 292
State formulas, 28–29
State holding element, 17
States, 13, 14–15. *See also* Locations
State transition system, 141
Static time-slicing, 253
String, as sequence of transitions, 160
Strongly connected components (SCCs), 36, 44, 129
Strong preservation, 232
Stubborn sets, 9
Stuttering equivalence, 146–154, 163–164
Subgroups, 215
Suffix, 29
Symbolic abstractions, 210–213
Symbolic model checking, xii, 61–95
  counterexamples, 71–74
  CTL, 66–87
  efficiency, 80–87
  example, 75–77, 85–87
  fairness, 68–71, 72–74, 91–95
  fixpoint representations, 61–66
  introduction to, 6–8
  LTL, 87–95
  partial order reduction, 143
  partitioned transition relations, 79–87
  real-time systems, 254, 255, 291
  relational product, 77–87
  symmetry, 228

Symbolic Model Verifier. *See* SMV
Symmetric groups, full, 215, 226, 228
Symmetry, 215–230
  complexity issues, 224–228
  example, 228–229
  group theory, 215–221
  introduction to, 10–11
  model checking methods, 221–223
  OBDDs, 222–223, 226–228
  orbit relation, 222–223, 226–228
  quotient models, 218–221, 222–223
Symmetry groups, types of, 226–228
Synchronous composition, 109
Synchronous execution, 17
Synchronous systems, 8, 18–19, 203–204
  abstractions, 193–194
  compositional reasoning, 187
  partitioned transition relations, 79
  real-time systems, 255, 265
Syntactic monotonicity, 98

Tableau construction, 42, 87–95
  ACTL formulas, 180–184, 186
  automata theory, 132–134
  compositional reasoning, 186, 190
Temporal analysis, quantitative, 256–264
Temporal logics, xiii, 27–49. *See also* ACTL; ACTL*; CTL; CTL*; LTL
  fairness, 32–33
  infinite families, 231–232
  introduction to, 4–6
  real-time systems, 254
Temporal operators, 27, 28
Terminal vertices, 51–52
Testing, method of, xi, xiii, 2
Theorem proving, xiii, 295
Tightening, of difference bound matrix, 288
Timed automata, 265–291
Time-slicing, static, 253
Timing constraints. *See* Quantitative temporal analysis
Timing diagram notation, 293
Token rings example, 232–233, 248–252
Topologies, 238, 294. *See also* Grammars; Graphs
Transitions, 13, 14–17. *See also* Partitioned transition relations
Transpositions, in group theory, 216
Trying region, 109, 111
Turing machine, 248–252

Undecidability, 231, 248–252
Unfolding technique, 9
Universal path quantifiers, 31
Universe of interpretation, 14
Unlock statement, 24
Until operator, 28

Valuation function, 14
Verification. *See also* Model checking
  methods of, xi, 1–3
  need for, xi-xii, 1–2
  probabilistic, 295
Vertices, in binary decision trees, 51–52
VERUS, 261–264
Visibility, 145, 162

Wait statement, 24
Weak preservation, 232
While statement, 22
Witnesses, 71–74. *See also* Counterexamples

Zone graph, 283–284, 289
Zones, 283. *See also* Clock zones