Symbolic Model Checking

An approach to the state explosion problem

Kenneth L. McMillan

May, 1992

CMU-CS-92-131

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

This research was supported in part by a scholarship grant from AT&T Bell Laboratories and in part by a National Science Foundation grant, no. CCR-8722633.

Keywords: formal verification, temporal logic, model checking, state explosion problem, symbolic model checking, binary decision diagrams, cache consistency, cache coherence, protocol verification, hardware description languages, SMV, asynchronous circuits, Petri nets, occurrence nets, process invariants.

Dedicated to the memory of William L. McMillan

Abstract

Finite state models of concurrent systems grow exponentially as the number of components of the system increases. This is known widely as the *state explosion problem* in automatic verification, and has limited finite state verification methods to small systems. To avoid this problem, a method called *symbolic model checking* is proposed and studied. This method avoids building a state graph by using Boolean formulas to represent sets and relations. A variety of properties characterized by least and greatest fixed points can be verified purely by manipulations of these formulas using Ordered Binary Decision Diagrams.

Theoretically, a structural class of sequential circuits is demonstrated whose transition relations can be represented by polynomial space OBDDs, though the number of states is exponential. This result is born out by experimental results on example circuits and systems. The most complex of these is the cache consistency protocol of a commercial distributed multiprocessor. The symbolic model checking technique revealed subtle errors in this protocol, resulting from complex execution sequences that would occur with very low probability in random simulation runs.

In order to model the cache protocol, a language was developed for describing sequential circuits and protocols at various levels of abstraction. This language has a synchronous dataflow semantics, but allows nondeterminism and supports interleaving processes with shared variables. A system called SMV can automatically verify programs in this language with respect to temporal logic formulas, using the symbolic model checking technique.

A technique for proving properties of inductively generated classes of finite state systems is also developed. The proof is checked automatically, but requires a user supplied process called a *process invariant* to act as an inductive hypothesis. An invariant is developed for the distributed cache protocol, allowing properties of systems with an arbitrary number of processors to be proved.

Finally, an alternative method is developed for avoiding the state explosion in the case of asynchronous control circuits. This technique is based the unfolding of Petri nets, and is used to check for hazards in a distributed mutual exclusion circuit.

Acknowledgments

To the extent that the work in this thesis is worthy of credit, Ed Clarke deserves a large portion of that credit, since his work is the foundation for almost everything in it. Ed got me started in the field by teaching me what I know about verification, and opening doors for me in the research community. His enthusiasm for my early, somewhat untutored tentatives in the field gave me the confidence to pursue my own ideas. Another important influence on this work and its author is Bob Kurshan. In more than once instance, his insistence on solving a particular problem led to a general solution in an unexpected way. What I now know about the process of research I learned from Ed Clarke and Bob Kurshan.

The work in this thesis also rests rather heavily on that of Randy Bryant. To a certain extent, the symbolic model checking technique resulted from the fortunate coincidence of my being at the same institution as Randy. Finding an application for this technique was also a more or less serendipitous occurrence, and for this I have to thank the people at Encore Computer Corporation, including Dyung Van Le, Jim Schwalbe and Drew Wilson, for taking an interest in formal verification, and generously allowing me to use their system as an application (and to publish it no less!).

I must also thank David Long and Jerry Burch, who contributed substantially to the ideas in this thesis. I owe special thanks to David Long, who is a font of information, and who helped me to solidify ideas in countless discussions, when no doubt he had more important things to do.

Thanks to Robert Brayton and the CAD group at the University of California at Berkeley, who gave this thesis a very careful and intelligent reading, and to Allan Fisher. It was always a pleasure to have discussions with Allan, and no doubt some of his insights can be found interspersed in these pages. Of course, I am indebted to all the good people who make the Carnegie Mellon School of Computer Science and its facilities work. CMU provides a unique environment for graduate students, and I consider myself privileged to have been a part of it.

Finally, to Tracy Slein, the one indispensible person in the whole process – I can't thank you enough.

Contents

Chapter 1

Introduction

There are several practical reasons for applying formal verification methods to computer systems. The most obvious is the high cost of correcting errors in digital designs. This cost has been increasing with the rising level of integration in digital circuit technology. It can be decreased to an extent in application specific designs by the use of programmable device technologies, but at least for the present, programmable logic has distinct disadvantages in performance and area. Thus, there is a growing demand for design methodologies that can yield correct designs on the first fabrication run. Design errors that are discovered before fabrication can also be quite costly, however, in terms of the engineering effort required to correct the error, and the resulting impact on development schedules. At present, the best tools available to engineers for finding errors before fabrication are simulators, which model the behavior of a system for predetermined or random input patterns. The engineer using simulation is faced with two ill-characterized and increasingly intractable problems. The first is creating a set of input patterns that are sufficient to expose any incorrect behavior of the system, and the second is determining the correct output of the system under these conditions, to be compared with the simulated output. Increased density of integration has allowed higher level functions such as network protocols to be implemented in hardware, and as a result, the problems of simulation have become critical. What seems to be needed is a precise yet understandable way of specifying correct behavior, and an exhaustive method of determining that the system model satisfies this specification for all input patterns. This is the meaning of formal verification.

A formal verification framework has three basic elements – a mathematical model of the system to be verified, a formal language for framing the correctness problem, and a methodology for proving the statement of correctness. One characteristic that many *automatic* verification methodologies have in common is that they require an exhaustive search of the state space of the model. Owing to simple combinatorics, the size of this state space can be, and usually is, exponential in the size of the system being modeled. This exponential growth in the state space, known as the *state explosion problem* is the limiting factor in applying automatic verification methodologies to large systems.

This thesis is directed toward solutions for the state explosion problem. This is essentially a question of methodology, but before we can discuss methodology, we need to discuss some of the models and formalisms that are commonly used in formal verification of hardware.

1.1 Background

The problem of hardware verification is in some ways similar to, and in other ways different from the problem of proving correctness of programs. Digital systems are most similar to what Pnueli has characterized as *reactive* programs [Pnu86], in that they receive input and produce output in a continuous interaction with their environment, rather than computing a single result and halting. In addition, the behavior of digital systems is concurrent in the extreme, since every gate in the system is simultaneously evaluating its output as a function of its inputs.

1.1.1 Temporal logic

For reasoning about concurrent, reactive programs, Pnueli proposed the use of a formal system originally studied by philosophers, called temporal logic [Pnu77, Pnu86, MP81, Kro87]. In a temporal logic, the usual operators of propositional logic are augmented by *tense operators*, which are used to form assertions about changes in time. One can assert, for example, that if proposition p holds in the present, then proposition q holds at some instant in the future, or at some instant in the past. The temporal modalities can be combined to express fairly complex statements about past, present and future. For example "if p holds in the present, then at some instant in the future, p will have held in the past." A temporal system provides a complete set of axioms and inference rules for proving all validities in the logic for a given model of time, such as partially ordered time, linearly ordered time, dense time, and even branching time.

Temporal logic is powerful enough to define a semantics for programs which captures not only the traditional before and after conditions of Floyd-Hoare style program proving, but also a wide variety of temporal properties of programs, such as termination, possible termination, termination under fair scheduling of concurrent processes, *etc.* [CE81a, BAMP81]. In the hardware area, Malachi and Owicki used temporal logic to give a concise specification of the conditions necessary for an asynchronous circuit to be *speed independent* [MO81]. Bochmann used temporal logic to give a semantics for self timed circuits, and used this system to verify a corrected version of an arbiter circuit [Sei80a]. Formal proofs of this kind are extremely tedious and difficult, however, and computationally intractable to automate. To simplify the hand proof, Bochmann used a somewhat oversimplified semantics for the circuit elements (neglecting gate delay) and as a result, missed a bug in the design, which was demonstrated by Dill [DC86].

A more practical application of temporal logic in hardware verification, called *model checking*, was introduced by Clarke and Emerson [CE81b] and independently by Quielle and Sifakis [QS81]. Instead of proving the validity of a logical formula for all models, a model checker determines the truth value of the formula in a specific finite model. For branching time logic, the model checking problem is computationally tractable, even though the validity problem is intractable. Here an important distinction between hardware and software systems comes into play – hardware systems are finite-state. This allows the proof procedure to be automated using model checking, while maintaining the formal elegance of temporal logic for specifying correct behavior.

The method of Clarke and Emerson first builds a complete state graph of the system from a description in an appropriate language.

The truth value of a formula in the logic is determined by an algorithm which propagates formulas in this state graph until a fixed point is reached. Besides being fast and fully automatic, this technique has the advantage that it can produce state sequences as counterexamples when the formula being checked is false. This has made it possible to find bugs in a number of small but fairly subtle circuit designs [BCDM86, BCD86], including the one verified by Bochmann.

For linear time logic, there is a decision procedure that translates a formula into an automaton by means of a *tableau* construction [RU71, CE81b, BAMP81]. This construction is similar the the semantic tableaux method of constructing proofs in standard logic [Smu68]. Each state in the tableau is associated with a set of formulas which are true in that state. Since the number of states in the tableau is exponential in the size of the formula, the method is not practical for proofs about very large systems. However, the tableau method can be used in a model checking framework, yielding an algorithm which is exponential in the size of the formula but linear in the size of the model [LP85].

1.1.2 Automata theoretic models

An alternative to the temporal logic framework is to cast the correctness problem in terms of a relation between the external or observable behaviors of two processes. One way to define this relation is by considering the set of all possible sequences of communications between processes. For example, in the L-automata model of Kurshan [Kur86], these sequences are defined by the language of an ω-automaton. Correctness is framed as the containment of the language of one automaton in the language of another. This asymmetric relation makes it possible to "underspecify" a system, that is, to leave some choices open to the designer. The use of automata on infinite strings makes it possible to express liveness properties. For instance, one can easily construct an automaton whose language is the set of all infinite strings such that every time a message is sent on some channel, one is eventually received. Language containment between ω-automata can be established by an algorithm which searches for cycles in the state space of a product automaton.

Van de Snepscheut [vdS83] and Dill [Dil88] have used trace theory to model speed independent circuits [Sei80b]. A trace is simply a history of the communications between a process and its environment. The trace sets of two process can be combined in a way which models communication between the two processes by synchronizing signals sent and received on the same channel. Dill's system is a circuit algebra which has both a structural interpretation (describing the physical connection of wires) and a trace theoretic interpretation (describing the communications along those wires). The actual trace sets are defined by the languages of finite automata (in this case, automata on finite strings, hence liveness cannot be modeled). A relationship called *conformance* between two processes determines when one process can safely be substituted for the other in all environments. Conformance can be tested by a polynomial algorithm which searches the state space of a finite automaton derived from the two processes.

In the Calculus of Communicating Systems (CCS) [Mil80], Milner takes a different approach in which external behavior is modeled by a tree rather than a set of sequences. The way CCS models communication is not well suited to modeling hardware, since in CCS a signal cannot be sent until a receiver is ready to receive it. In hardware, a receiver cannot generally prevent a signal from being sent. Also, in CCS, communication is always between two processes, while in hardware signals are often broadcast to many receivers. A calculus specialized to circuits called CIRCAL [Mil83] was developed to remedy these problems. The notion of correctness in process calculi is called *observational* equivalence, meaning that an observer cannot distinguish between two processes by any experiment. This notion of correctness is extremely strict, since it doesn't allow the specifier to leave any choice up to the designer regarding the externally visible behaviors. Observational equivalence can be proved by establishing a relation called *bisimulation* between the two processes. For finite state processes, there is an polynomial time algorithm for bisimulation which is very similar to the coarsest partitioning algorithms used for state machine minimization [NH84].

All of these methods can be viewed as variations on the theory of finite automata, tailored for modeling certain properties of a particular class of systems. In fact, the automata theoretic approach is not very far from the temporal logic approach in practice. The difference is mostly a question of notations, since the *tableau* method provides a way of translating a temporal logic formula into an automaton. Although temporal logic is not as expressive as automata in characterizing classes of sequences, it has been shown by Wolper that temporal logic can be extended using right linear grammars to make it as expressive as automata without increasing the complexity of the decision procedure [Wol83]. Clarke and Kurshan have also proposed a branching time logic in which the temporal operators are defined by finite ω-automata [CGK89].

What all of the above systems have in common is that correctness, once formalized, can be determined by an algorithm that searches the entire state space of a finite state model. Such methods have the advantage of being fully automatic, but invariably suffer from the state explosion problem.

1.2 Scope of the thesis

This thesis explores methods of state space search that avoid the state explosion problem by not explicitly representing the states of the model. To do this, some revolutionary new techniques are borrowed from the area of switching function analysis. In this domain, a combinational explosion also arises, since the number of input combinations to a Boolean function is exponential in the number of inputs. New techniques for Boolean comparison avoid this problem by representing Boolean functions with a reduced form of decision graph called an Ordered Binary Decision Diagram (OBDD) [Bry86]. These decision graphs provide a compact canonical form for Boolean functions. To apply this idea to temporal verification, we observe that if the state of a system is represented by a vector of Boolean variables, then a set of states can be represented by a Boolean function which returns true for all states in the set. Similarly, a relation xRy between states can be represented by a Boolean function of two sets of variables, one set representing x and the other representing y. In this way, a model checking algorithm can be developed which uses OBDDs to represent sets and relations. Borrowing terminology from Bryant, this technique is called *symbolic* model checking, since symbolic variables are used to represent the components of the system state rather than numeric values. Using symbolic model checking, we can can automatically verify some regularly structured systems with literally astronomical numbers of states.

The principle contributions of this work are detailed below:

The symbolic model checking method. A technique is developed for state space search using Ordered Binary Decision Diagrams. We show that any algorithm that can be expressed in a fixed point logic called the Mu-Calculus can be computed using this method. These include algorithms for all of the correctness notions enumerated above, including CTL model checking (with fairness constraints), the linear time tableau method, conformance, observational equivalence, language containment for ω-automata, Mealy machine equivalence, and others. From a theoretical point of view, a structural class of sequential circuits is identified whose transition relations can be represented by a polynomially bounded OBDDs. This theoretical result is born out by experiments on classes of regularly structured circuits, for which the time used by the symbolic model checking method is found to be polynomially bounded in the circuit size. In addition, some experiments are reported using symbolic model checking to compute the equivalence relation between states of a finite state machine. Several techniques are advanced which improve the efficiency of this computation in practice.

The SMV system. A symbolic model checking system called *SMV* is presented. This system permits the automatic verification of programs written in a specialized language for describing concurrent finite state systems and protocols. This language is somewhat similar to LUSTRE [CHPP87] in its synchronous dataflow semantics, but has several unique aspects. For example, it allows systems to be modeled *nondeterministically* for purposes of abstraction, it allows arbitrary interleaving of concurrent processes, and it allows programs to be annotated with assertions in branching time temporal logic.

Formal verification of the Encore Gigamax cache consistency protocol. The cache consistency protocol of a distributed shared-memory multiprocessor called the Encore Gigamax is modeled in the SMV language and verified using the symbolic model checker. Running in minutes, the symbolic model checker discovered errors in this system which were not discovered by simulation, in spite of the very large state space of the model [MS91]. This experiment shows that the model checking technique can be used effectively in an industrial setting for highly complex systems. It also sheds light on issues involved in modeling such protocols as finite state systems, and the kinds of errors that can be found by model checking that are not likely to be found by simulation.

Induction over processes. A partially automated method of induction is described for proving properties of parameterized classes of designs. The method applies to a variety of process models, requiring of the model only certain simple algebraic properties. The SMV system is extended to support proof by induction, allowing some properties of the Gigamax cache protocol to be verified for configurations of arbitrary size.

Verification using occurrence nets. An alternative method for avoiding the state explosion is examined. This technique avoids considering all of the possible interleavings of concurrent actions by using a partially ordered representation of behavior called an occurrence net [NPW81]. This method is used to verify that a design for an asynchronous distributed mutual exclusion circuit is hazard free (this example is also used for the symbolic model checking method). Using this technique, we also find empirically that the run time is polynomial in the number of components of the system, while the number of states is exponential.

1.3 Related research

Since the state explosion problem is ubiquitous in the verification of computer systems and protocols, many researchers in the area have studied it.

1.3.1 Reduction

The most common approach is based on *reduction* – reducing the correctness problem to a similar problem in a smaller state space. This is generally done by replacing processes in the model by smaller processes that have similar or identical communication behavior. The most general framework for this kind of reduction is that of Kurshan [Kur87]. Using *homomorphic reductions* of ω-automaton models, it is possible to simplify not only the internal state of a process, but also its external communications. In this methodology, one generally builds a hierarchy of reductions, in which processes at the lowest level are reduced, then combined at the next higher level and further reduced, *etc.* Kurshan advocates building this hierarchy from the top down, so that the most abstract models can be verified before details are filled in at the next lower level.

A hierarchical approach was also taken by Dill in his trace theoretic system for speed independent circuits [Dil88]. In this case, the reduction is obtained mostly by hiding internal signals of a module. There is no provision for abstracting the signals by which the module communicates with its environment. That is, communication always remains at the same level, that of digital signal transitions.

The reduction approach is generally not automatic. Usually, the reduced process is obtained in an *ad hoc* manner, and the validity of the reduction is then tested automatically. Some methods have been proposed for obtaining reduced processes automatically, however. For example, in a method called *compositional model checking*, a state minimization procedure is used to obtain a reduced process that is equivalent to the original process with respect to observation via its inputs and outputs [CLM89b, CLM89a]. This reduction preserves the truth value of all formulas in a suitable logic. Graf and Steffen have also studied minimization with respect to a suitable notion of equivalence as a reduction technique [GS91]. Minimization techniques are fairly strict in terms of the required relation between the original and reduced processes, however. As a result, the reduction that can be obtained using these techniques is not generally as great can be obtained using more flexible but unautomated methods.

The symbolic model checking technique is not really an alternative to reduction methods, but is complementary to them. In general, the larger the state space that can be searched automatically, the less the need for reduction. For example, Dill used a reduction (constructed by hand) to verify a speed independent distributed mutual exclusion ring circuit [Dil88]. Using symbolic model checking, there is no need for a reduction – the verification time is polynomial in the size of the ring (cf. chapter 2). On the other hand, symbolic model checking techniques can be used to implement the validity test for reductions (cf. chapter 5), hence the two techniques can be combined.

1.3.2 Induction

In systems of many identical processes, it is sometimes possible to reduce an arbitrary number of processes to a single process while retaining certain properties of interest. For example, Browne, Clarke and Grumberg proposed a reduction technique of this sort which preserves the truth value of formulas in a suitably restricted logic with process quantifiers [BCG86]. Unfortunately, the reduction, a form of bisimulation, had to be established by hand. There was no automated way of checking it. Kurshan and McMillan proposed an inductive method of establishing the reduction that could be checked automatically [KM89]. The method is also less restrictive in terms of the properties that can be proved since it does not rely on bisimulation. This method is used in chapter 5. A similar method was described independently by Wolper and Lovinfosse [WL89]. Another inductive technique has been described by Shtadler and Grumberg [SG89]. This technique is somewhat more flexible in that it treats networks generated by context free grammars, but is limited to bisimulation as a reduction technique. A more detailed comparison of these methods can be found in chapter 5.

1.3.3 Other symbolic methods

Coudert and Madre have described a method for verifying finite state machines using Ordered Binary Decision Diagrams which is similar to symbolic model checking [CBM89]. The symbolic model checking technique was developed in 1987. The technique of Coudert and Madre appears to have been developed two years later [Cou91] but independently. There are several differences of approach between the two methods. Symbolic model checking is directed mostly toward proving temporal properties of finite state systems, whereas Coudert and Madre have concentrated mostly on proving equivalence of deterministic Mealy machines (though they also discuss temporal logic [CMB91]). Testing Mealy machine equivalence is useful, for example, when one is mapping a design from one technology to another, but is a fairly limited form of verification, since the specification is at the same level of detail as the implementation. Also, in this work, we consider the performance of algorithms mostly in terms of asymptotic behavior for regularly structured classes of systems, while Coudert and Madre have considered mostly a set of benchmark circuits for synthesis. This makes it difficult to determine how well their technique scales with circuit size. Finally, Coudert and Madre have not as yet reported any results for testing equivalence of two different implementations of the same finite state machine. In practice, they have only used symbolic techniques to generate the set of reachable states of a finite state machine. This information is useful for test generation and sequential synthesis [TSL$^+$90], but these experiments provide no information about how well the technique works for verification. On the other hand, the symbolic model checking technique has been applied to the verification of an industrial design for a distributed cache consistency protocol (cf. chapter 4). A more detailed description of the work of Coudert and Madre, and others using OBDDs for sequential circuit verification, can be found in chapter 2.

Chapter 2

Symbolic model checking

As mentioned in the introduction, a formal verification system has several basic elements. First, we require a *model*. A model is an imaginary universe, or more generally, a class of possible imaginary universes. To make our model meaningful, we require a *theory* that predicts some or all of the possible observations that might be made of the model. An observation generally takes the form of the truth or falsehood of a predicate, or statement about the model. Finally, to verify something meaningful about the model, we require a *methodology* for proving statements that are true in the theory.

In program proving, the universe is a totally imaginary one, driven by mechanisms (the compiler and hardware) of which the programmer has no knowledge. The logician is free to assign any semantics at all to programs, provided the compiler writer and hardware designer agree to implement them. This makes program proving an artificial science, in the sense that our theory is true because we say it is. In contrast, a hardware verification system requires a model of a real physical system. The underlying physical mechanism is still invisible to us (we can only postulate its existence), but we can empirically construct a model which predicts the necessary observations with a sufficient degree of accuracy for our purposes (the verification of digital circuits). It turns out that the required degree of accuracy is not very large. Though quite accurate models are possible, using partial differential equations to describe the time evolution of fields and particle densities, a suitable design style makes it possible to consider only the digital (one or zero) value of voltages, ignoring entirely the exact voltage within the digital ranges, and the time it takes to switch from one range to another. Depending on the design style (*eg.*, synchronous or self timed), different models may be appropriate. In certain rare cases, we may have to use differential equations to model the analog behavior of circuits (for example, when metastability arises). In this thesis, though, we will consider only fairly abstract models of circuits as finite state machines. Thus, we return to the science of the artificial, wherein we choose the theory to suit our needs, but with the understanding that a method exists for translating our models into real systems.

The kind of theory that emerges for the model depends to a large extent on the kind of experiments the observer is able to perform. For example, in traditional program proving systems, the observer is allowed to set up the initial state of the program, wait for the program to terminate, and then examine the final state. The theory of this model can be expressed in a kind of before-and-after logic whose axioms determine the semantics of programs. For example, in Floyd-Hoare logic [Hoa69], the formula

$$\{\text{true}\}\ x := y\ \{x = y\}$$

is an axiom: for any initial condition, after the program $x := y$ terminates, x and y have the same value. The fact that no other variables change value in the process can also be expressed as an axiom:

$$\{z = a\}\ x := y\ \{z = a\}$$

provided neither z nor a depend on x.

In this system, if the program fails to terminate (diverges), the observer must simply wait forever, *ie.*, no observation is possible. One might ask whether waiting forever is not itself an observation, that is, should it not be possible to state in the semantics that a given program terminates or doesn't terminate for a given initial condition? This point can be argued either way for programs (since knowing that a program terminates before infinity is not very practical information). However, for digital systems (or reactive systems in general), it is clear that simple before and after conditions are not a sufficient theory; first of all termination for these systems is not well defined, and moreover the meaning of what these systems are supposed to *do* is inseparably linked with the evolution of events in time [Pnu77].[1] What we need is a formal theory in which we can reason about temporal aspects of a system's behavior.

2.1 Temporal logic

Temporal logic (or *tense* logic) is a system devised by philosophers expressly for making statements about changes in time [Bur84]. In temporal logic, the formula Fq is true in the present if q is true at some moment in the future. Similarly Pq is true in the present if q is true at some moment in the past. These tense operators, F and P, have duals which are generally given their own names. The formula Gq is equivalent to $\neg F \neg q$, meaning that q is true at every moment in the future. The formula Hq is equivalent to $\neg P \neg q$, meaning that q is true at every moment in the past. These operators can give surprisingly concise expressions of sentences with complex tense structures. For example, $q \Rightarrow FPq$ can be interpreted as "if q holds in the present, then at some time in the future q will have held in the past".

The usual model theoretic semantics given to temporal logic (and other *modal* logics) is the so-called *possible worlds* semantics. A *frame* in this semantics consists of a class S of states through which the system evolves, and a relation $<$ representing temporal order. A *model* is a frame with a *valuation* L, which assigns truth or falsehood to every atomic proposition (propositional letter) in every state.[2] The truth or falsehood of temporal formulas is relative to the present state. For example, the formula Fq is true in state s iff there exists a state t such that p is true in state t and $s < t$. Similarly, Pq is true in state s iff there exists a state t such that p is true in state t and $t < s$. Notice that a temporal formula acts like an open sentence, with one free parameter representing the present state. Thus it defines a class of states in which the formula is true. Similarly, a state defines a class of formulas which are true in that state.

[1] The question of termination is in any event not undecidable for hardware systems, since they are not computation universal (only programs are).

[2] These are usually called Kripke frames and Kripke models, after one of the first mathematicians to give a model theoretic interpretation of modal logic.

The choice of axioms in the logic can be used to characterize the temporal ordering relation $<$. For example, the following axioms (in addition to the propositional tautologies) exactly characterize those frames whose $<$ relation is a partial order (transitive and antisymmetric) [Bur84]:

$$G(p \Rightarrow q) \Rightarrow (Gp \Rightarrow Gq) \tag{2.1}$$

$$H(p \Rightarrow q) \Rightarrow (Hp \Rightarrow Hq) \tag{2.2}$$

$$p \Rightarrow GPp \tag{2.3}$$

$$p \Rightarrow HFp \tag{2.4}$$

One inference rule (in addition to *modus ponens*) is required: by *temporal generalization*, if α is provable, we infer that $G\alpha$ and $H\alpha$ (that is, a tautology must hold true at all times, or perhaps, the rules of sound inference do not change with time). By specializing this system slightly, we can obtain logics characterizing a variety of models of time, including linear time, discrete time, and branching (non-deterministic) time. All of these results can be found in [Bur84].

2.1.1 Linear time

We usually think of time as a linearly ordered set, measuring it either with the real numbers or the natural numbers. A frame is linearly ordered if, in addition to being partially ordered, it is total, *ie.*, for all states s, t, either $s < t$, $s = t$, or $t < s$. The temporal frames in which $<$ is a linear order can be characterized by simply adding the following two axioms to the basic set (they are time reversal duals):

$$(FPq) \Rightarrow (Pq \vee q \vee Fq) \tag{2.5}$$

$$(PFq) \Rightarrow (Pq \vee q \vee Fq) \tag{2.6}$$

Linear temporal logic is usually extended by the *until* operator and the *since* operator. Informally, $p\ U\ q$ states that p will hold at some moment in the future, until which time q will hold at all moments. Similarly, $p\ S\ q$ states that p held at some moment in the past, since which time q has held at all moments. More precisely, $p\ U\ q$ is true in state s if there is some state t such that $s < t$ and q is true in state t, and for all $s < u < t$, p is true in state u.

2.1.2 Discrete time

It is common in engineering to model time as a discrete sequence (measured by the integers). Discrete dynamics are commonly used, for example, in signal processing and synchronous digital systems. A discrete frame is one in which every state has an immediate successor and an immediate predecessor. The linear discrete frames can be characterized by adding the following two axioms to those for linear time logic:

$$p \wedge Hp \Rightarrow FHp \tag{2.7}$$

$$p \wedge Gp \Rightarrow PGp \tag{2.8}$$

It is useful in a discrete linear temporal logic to define a *next time* temporal operator. The formula Xq is true in state s when there is an immediate successor of x in which q is true. A state t is an immediate successor of s if $s < t$ and there does not exist a state u such that $s < u < t$. Thus, Xq is exactly equivalent to false $U\ q$, so its addition does not increase the expressiveness of the logic.

2.1.3 Branching time

A branching frame is one in which the temporal order $<$ defines a tree which branches toward the future. Thus, every instant has a unique past, but an indeterminate future. This is an inherently non-deterministic model of time, and hence is well suited, for example, for defining a semantics of non-deterministic programs. A frame is tree ordered when for all states s, t, u, if $t < s$ and $u < s$ then $t < u$, $t = u$ or $t > u$. In other words, the past of every state is linearly ordered. The tree ordered frames can be characterized by simply dropping (2.6) from the axioms of linear time logic.

Though pure tense logic can exactly characterize the branching time frames, it leaves something to be desired in expressing properties of non-deterministic programs. For example, it is common in defining the semantics of these programs to say that a program aborts iff it must inevitably abort. This functionality can be implemented by backtracking. Similarly, a non-deterministic Turing machine terminates if it may possibly terminate. These notions of inevitability and possibility are not represented in an ordinary tense logic. They can be incorporated, however, by combining notions from temporal logic and modal logic.

We would like to interpret the branching structure of time as meaning that each instant of time has many possible futures, and that as time evolves from present to future, these possibilities are reduced. Thus, in the past, there existed possible futures which are now precluded. This interpretation gives rise to notions of necessity (inevitability) and possibility in tense logic [Tho84]. We think of the truth or falsehood of tense formulas as being relative to a given branch of the tree ordered frame (one possible evolution of time into the future). A branch is defined as a maximal linearly ordered set of states. We will write $q[s, b]$ if q holds in state s in branch b. Thus, $Fq[s, b]$ iff there exists a state t in b such that $s < t$ and $q[t, b]$. Similarly, $Pq[s, b]$ iff there exists a state t in b such that $t < s$ and $q[t, b]$. The notion that q is *necessarily* true is represented by the formula Aq. We will say $Aq[s, b]$ iff for all branches b' containing s, $q[s, b']$. The notion that q is *possibly* true is represented by the formula Eq. We will say $Eq[s, b]$ iff for some branch b' containing s, $q[s, b']$. Notice that A and E provide a kind of second order quantification over maximal linearly ordered subsets.[3]

According to this semantics for modal branching time logic, there may be possibilities in the past that are foreclosed in the present. For example, $q \Rightarrow HAFq$ is not valid. The fact of q in the present does not imply the necessity of q in the past. Thus, modal branching time logic might be termed the logic of regret. The logic can also express useful semantic properties of non-deterministic programs [BAMP81]. For example, if q represents the fact of a program terminating, then inevitable termination is expressed by the formula AFq (necessarily in the future q). Possible termination is expressed by EFq (possibly in the future q). If the proposition p represents a correct output of the program, then (inevitable) partial correctness is expressed by the formula $AG(q \Rightarrow p)$ (necessarily invariantly, termination implies correctness). The somewhat odd but definable notion of possible partial correctness is expressed by $EG(q \Rightarrow p)$. Note that Pq, APq and EPq are all logically equivalent, since the past of a state is the same for any branch.

[3]Classically, the symbol $\Box$ is used to represent necessity, and $\Diamond$ is used to represent possibility. The symbols A and E are used here for consistency with [BAMP81].

Also note that A and E are dual, since Aq is equivalent to $\neg E \neg q$.

2.2 The temporal logic CTL

The temporal logic CTL is a subset of modal branching time logic defined by Clarke and Emerson [CE81b]. The acronym stands for *Computation Tree Logic.*[4] In CTL, temporal operators occur only in pairs consisting of A or E, followed by F, G, U or X. Thus, past time operators are not allowed, and tense operators cannot be combined directly with the propositional connectives.

2.2.1 Syntax and semantics of CTL

The syntax of CTL formulas is given as follows:

1. Every atomic proposition is a CTL formula.
2. If f and g are CTL formulas, then so are

$$\neg f,\ (f \wedge g),\ AXf,\ EXf,\ A(fUg),\ E(fUg)$$

The remaining operators are viewed as being derived from these according to the following rules:

$$\begin{aligned} f \vee g &= \neg(\neg f \wedge \neg g) \\ AFg &= A(\text{true } U\ g) \\ EFg &= E(\text{true } U\ g) \\ AGf &= \neg E(\text{true } U\ \neg f) \\ EGf &= \neg A(\text{true } U\ \neg f) \end{aligned}$$

The truth or falsehood of formulas is defined with respect to a Kripke model, but in a slightly non-standard way. For CTL, the model is a triple (S, R, L), where S is the set of states, R is the *transition relation* and L is the valuation. The transition relation is the set of

[4]CTL is actually a subset of a more general temporal logic described in [CE81a], adopting the syntax of [BAMP81].

all pairs (s, t) such that t is an *immediate successor* of s. A branching model (*a.k.a.* computation tree) can be obtained by starting at a designated state s and unwinding the graph (S, R) into an infinite tree (provided every state has at least one successor). The semantics for CTL given below is equivalent to the standard semantics with respect to this infinite tree.[5]

A *path* of a model $K = (S, R, L)$ is an infinite sequence of states $(s_0, s_1, s_2 \ldots) \in S^\omega$ such that each successive pair of states (s_i, s_{i+1}) is an element of R. Every path is maximal linearly ordered subset of the tree structure unwound from s_0.

The notation $K, s \models f$ means that the formula f is true in state s of Kripke model K. In the sequel, where the model is unambiguous, we will write simply $s \models f$. The interpretation of a CTL formula f with respect to a Kripke model K is given below, by recursion over the structure of formulas:

$$\begin{array}{rcl}
s \models p & \text{iff} & L(s)(p), \text{ where } p \text{ is an atomic proposition} \\
s \models \neg f & \text{iff} & s \not\models f \\
s \models f \wedge g & \text{iff} & s \models f \text{ and } s \models g \\
s_0 \models AXf & \text{iff} & \text{for all paths } (s_0, s_1, \ldots),\ s_1 \models f \\
s_0 \models EXf & \text{iff} & \text{for some path } (s_0, s_1, \ldots),\ s_1 \models f \\
s_0 \models A(f\ U\ g) & \text{iff} & \text{for all paths } (s_0, s_1, \ldots), \text{ for some } i, \\
 & & \quad s_i \models g \text{ and} \\
 & & \quad \text{for all } j < i,\ s_j \models f \\
s_0 \models E(f\ U\ g) & \text{iff} & \text{for some path } (s_0, s_1, \ldots), \text{ for some } i, \\
 & & \quad s_i \models g \text{ and} \\
 & & \quad \text{for all } j < i,\ s_j \models f
\end{array}$$

2.2.2 Fixed point characterization of CTL

Emerson and Clarke [CE81a] showed that various branching time properties of programs can be characterized as extremal fixed points of appropriate continuous functionals. Later, they introduced the logic CTL, and showed that its operators can be characterized in this way [CE81b]. This characterization led to an efficient algorithm for the model checking problem – determining whether a given CTL formula is satisfied in

[5]With one additional distinction: in CTL, the future is taken to include the present. Thus, if p holds in the present, then so does Fp.

a given state of a finite Kripke model.

To obtain the fixed point characterization, we will identify each CTL formula f with $\{s \mid s \models f\}$, the set of states in which the formula is true. In this way, for example, true denotes the empty set, false denotes S, and every subset of S represents an equivalence class of formulas.[6] Let $\mathcal{P}(S)$ be the set of subsets of S. $\mathcal{P}(S)$ forms a lattice under union and intersection. This lattice is ordered by set inclusion, where $P \subseteq Q$ if and only if $P \cup Q = Q$. A functional $\tau[Y]$ is a formula with one uninterpreted propositional letter Y. This defines a function $\mathcal{P}(S) \rightarrow \mathcal{P}(S)$, where $\tau(P)$ is obtained by taking P for Y in τ. By definition:

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

When the set S is finite, every increasing chain of subsets has a maximum element, and every decreasing chain has a minimum element. As a result, in the finite case, monotonicity implies both $\cup$-continuity and $\cap$-continuity.

A fixed point of τ is any P such that $\tau(P) = P$. Tarski [Tar55] showed that a monotonic functional always has a least and a greatest fixed point with respect to inclusion ordering:

Theorem 1 (Tarski-Knaster) *Whenever $\tau[Y]$ is monotonic, it has a least fixed point, denoted $\mu Y.\tau[Y]$ and a greatest fixed point, denoted $\nu Y.\tau[Y]$. When $\tau[Y]$ is also $\cup$-continuous, $\mu Y.\tau[Y] = \cup_{i\geq 0}\tau^i(\text{false})$. When $\tau[Y]$ is also $\cap$-continuous, $\nu Y.\tau[Y] = \cap_{i\geq 0}\tau^i(\text{true})$.*

We can now characterize the CTL operators in terms of fixed points of appropriate functionals:

Theorem 2 (Clarke-Emerson) *Provided S is finite,*

[6]This is essentially an algebraic interpretation of logic, where we embed the formulas of the logic in a Boolean algebra $(\mathcal{P}(S), 0, 1, \cap, \cup, -)$, with $\cap$ representing conjunction, $\cup$ representing disjunction and $-$ (set complement) representing negation

1. $EFp = \mu Y.(p \vee EXY)$
2. $EGp = \nu Y.(p \wedge EXY)$
3. $E(q\ U\ p) = \mu Y.(p \vee (q \wedge EXY))$

There is a standard algorithm for computing the least [greatest] fixed point of a monotonic functional. This is done by starting with false [true] and iterating the functional until a fixed point is reached, as shown below. Assuming S is finite, this procedure terminates in at most $|S| + 1$ iterations with the least [greatest] fixed point of $\tau[Y]$:

```
to compute $\mu Y.\tau[Y]$ {or $\nu Y.\tau[Y]$} :
    let $Y$ = false; {or $Y$ = true}
    do
        let $Y' = Y$, $Y = \tau[Y]$
    until $Y' = Y$;
    return $Y$
```

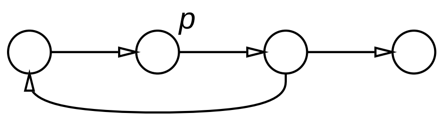
Theorem 3 *Given a finite set S, and a monotonic functional $\tau[Y]$, the standard fixed point algorithm computes $\mu Y.\tau[Y]$ {or $\nu Y.\tau[Y]$} in at most $|S| + 1$ iterations.*

Proof. Since τ is monotonic, $\tau^0[\text{false}] \subseteq \tau^1[\text{false}] \subseteq \tau^2[\text{false}] \cdots$. The longest *strictly* increasing chain of subsets of S has length $|S| + 1$. Hence, there must be an i such that $0 \leq i \leq |S|$ and $\tau^i[\text{false}] = \tau^{i+1}[\text{false}]$ (otherwise there would be a strictly increasing chain of length $|S| + 2$). Hence, the algorithm terminates after at most $|S| + 1$ iterations. For any such i, $\cup_{j \geq 0} \tau^j[\text{false}] = \tau^i[\text{false}]$. Hence, by theorem 1, $\mu Y.\tau[Y] = \tau^i[\text{false}]$.

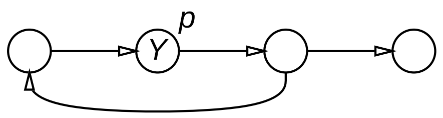
For the greatest fixed point, substitute true for false, $\supseteq$ for $\subseteq$, and decreasing for increasing in the above argument. □

Having a fixed point characterization of the CTL operators allows us to use the standard fixed point algorithm to determine the set of states of a given model in which a CTL formula is true. As an example, consider computing EFp in the following Kripke model:[7]

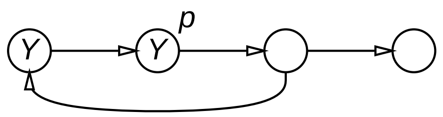
[7]We represent a Kripke model pictorially by drawing the graph (S, R) and labeling each state with the atomic propositions which are true in that state.



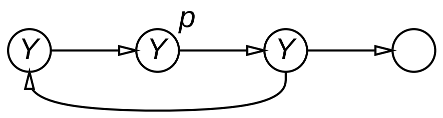
Since $|S| = 4$, the number of iterations required to produce the fixed point is at most 4. Therefore, let us compute $\tau^i[\text{false}]$ for $i = 1 \ldots 4$, where $\tau[Y] = p \vee EXY$. After the first iteration, we have $\tau^1[\text{false}] = p \vee EX\text{false} = p$:



After the second iteration, we have $\tau^2[\text{false}] = p \vee EXp$:

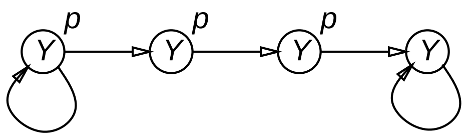


After the third iteration, we have $\tau^3[\text{false}] = p \vee EX(p \vee EXp)$:

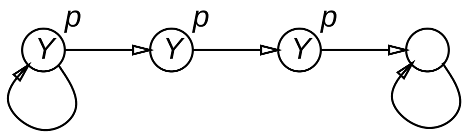


which is a fixed point, since the next iteration, $\tau^4[\text{false}]$ produces the same result. Notice that at each iteration i, we have the set of states s_0 such that there exists a path $(s_0, s_1, s_2, \ldots)$ where p is true at some state less than i. This algorithm can be thought of as a backward breadth first search of the graph. In the end, we have labeled exactly the set of states on a path to a state labeled with p.

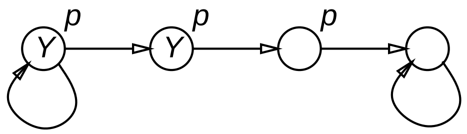
As a second example, consider computing EGp in the following Kripke model:



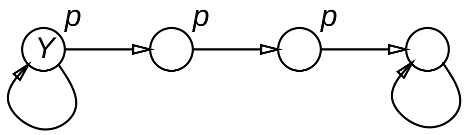
After the first iteration, we have $\tau^1[\text{true}] = p \wedge EX\text{true} = p$:



After the second iteration, we have $\tau^2[\text{true}] = p \wedge EXp$:



After the third iteration, we have $\tau^3[\text{true}] = p \wedge EX(p \wedge EXp)$:



This is the greatest fixed point, since the next iteration $\tau^4[\text{true}]$ produces the same result. Notice that at iteration i, we have the set of states such that there exists a path of length i where every state satisfies p. When we reach a fixed point, every state in the set has a successor in the set satisfying p, hence for every state in the set, there exists an infinite path where p is always true.

The operators EX, $E(\ U\ )$ and EG are actually sufficient to characterize the entire logic, since the remaining operators can be derived from these three according to the following rules:

$$\begin{aligned} EFp &= E(\text{true}\ U\ p) \\ AXp &= \neg EX \neg p \\ AGp &= \neg EF \neg p \\ A(q\ U\ p) &= \neg(E(\neg p\ U\ \neg q \wedge \neg p) \vee EG \neg p) \end{aligned}$$

For this reason, in the sequel, we will consider only the operators EX, $E(\ U\ )$ and EG. However, for completeness, here are the fixed point characterizations of the remaining operators:

$$\begin{aligned} AGp &= \nu Y.(p \wedge AXY) \\ A(q\ U\ p) &= \mu Y.p \vee (q \wedge AXY) \end{aligned}$$

The fixed point characterization provides an effective algorithm for the model checking problem. In fact, a more efficient algorithm exists, based on breadth first search and the calculation of strongly connected components in the graph (S, R) [CES86]. Both of these algorithms suffer from the state explosion problem, however; it is necessary to construct the complete state graph of the system being modeled before model checking can be applied. Since the number of states of a system grows exponentially in the number of its components, these algorithms can only be applied to small systems.

2.3 Symbolic CTL model checking

In the previous section, we equated a CTL formula with the set of states in which the formula is true. We showed how the CTL operators can thus be characterized as fixed points of certain monotonic functionals in the lattice of subsets, and how these fixed points can be computed iteratively. In this section, we equate sets and relations with Boolean formulas, and show how set theoretic operations such as union, intersection and image can be characterized in terms of Boolean operations. This allows the CTL model checking algorithm to be implemented using well developed automatic techniques for manipulating Boolean formulas. Since the state graph is symbolically represented by a Boolean formula, there is no need to actually construct it as an explicit data structure. Hence, the state explosion problem can be avoided.

2.3.1 Quantified Boolean formulas

Quantified Boolean Formulas (QBF) are an extension of propositional logic allowing quantifiers over propositional variables. Given a set V of propositional variables, QBF(V) is the least set of formulas such that

1. true and false are formulas,
2. every variable in V is a formula,
3. if p and q are formulas, then so are $p \vee q$ and $\neg p$, and
4. if p is a formula and v is in V, then $\exists v.\ p$ is a formula.

A truth assignment is a function $V \to \{\text{false}, \text{true}\}$. We equate each QBF formula with the set of truth assignments that satisfy the formula. Thus, true represents the set of all truth assignments, false the empty set, and a propositional variable v represents the set of all truth assignments a such that $a(v) = \text{true}$. In addition,

1. $a \in (p \vee q)$ if and only if $a \in p$ or $a \in q$,

2. $a \in (\neg p)$ if and only if $a \notin p$, and

3. $a \in (\exists v.\ p)$ iff $a(v \leftarrow \text{true}) \in p$ or $a(v \leftarrow \text{false}) \in p$.

It is useful to define an operator for QBF that substitutes a formula for a variable. If p and q are QBF formulas, and v is a variable, then let $a \in p(v \leftarrow q)$ if and only if $a(v \leftarrow (a \in q)) \in p$. Note that quantification can be defined in terms of substitution, since $\exists v.\ p = p(v \leftarrow \text{false}) \vee p(v \leftarrow \text{true})$.

Quantification and substitution can also be defined for vectors of variables. If $W = (w_1, \ldots, w_k)$ is an n-tuple of propositional variables and $Q = (q_1, \ldots, q_n)$ an n-tuple of formulas, then let

1. $a \in \exists W.\ p$ iff for some $b : W \to B$, $a(w_i \leftarrow b(w_i)) \in p$ and

2. $a \in p(W \leftarrow Q)$ iff $a(w_i \leftarrow (a \in q_i)) \in p$.

2.3.2 Representing sets and relations

The state of a concurrent system is generally modeled as vector where each component represents the state of one component of the system. For the moment, let us make the simplifying assumption that all of the state components are Boolean valued, as is generally the case in digital systems. A state of the system can therefore be viewed as a truth assignment to a set of propositional variables $V = \{v_1, v_2, \ldots, v_n\}$. Under this interpretation, every QBF formula over the set of state variables V denotes a set of states, *ie.*, the set of truth assignments which satisfy the formula. For example, if we have two state variables a and b, then the formula $a \vee b$ represents all the states in which a is true or b is true.

In order to represent a binary relation with a QBF formula, we introduce two ordered sets of variables $V = \{v_1, \ldots, v_2\}$ and $V' = \{v'_1, \ldots, v'_2\}$. The set V represents the left argument of the relation, and the set V' represents the right argument. By this arrangement, a QBF formula R over the variables $V \cup V'$ stands for a binary relation R', the set of pairs (x, y) in $(V \to B)^2$ such that

$$(x, y) \in R' \text{ iff } x(v'_i \leftarrow y(v_i)) \in R \tag{2.9}$$

As an example, if we have two state variables, a and b, then the QBF formula $a \wedge b'$ represents all ordered pairs of states such that a is true in the first state, and b is true in the second state.

Using this representation, we can express a variety of standard set theoretic operations in terms of the QBF connectives. For example, the union of two sets represented by A and B is $A \vee B$, their intersection is $A \wedge B$ and the complement of A is $\neg A$.

The image $R'(Q)$ of a set Q via a binary relation R' is the set of all y such that for some $x \in Q$, $(x, y) \in R'$. If R is a QBF formula representing a relation R', and Q is a QBF formula representing a set, then

$$R'(Q) = (\exists V.\ (R \wedge Q))(V' \leftarrow V) \tag{2.10}$$

We can prove this by simply expanding the definitions of the QBF operators, as follows:

$$\begin{gathered}
y \in (\exists V.\ (R \wedge Q))(V' \leftarrow V) \\
\text{iff} \\
y(v'_i \leftarrow y(v_i)) \in \exists V.\ (R \wedge Q) \\
\text{iff} \\
\text{exists } x : V \to B \text{ s.t. } y(v'_i \leftarrow y(v_i))(v_i \leftarrow x(v_i)) \in R \wedge Q \\
\text{iff} \\
\text{exists } x : V \to B \text{ s.t. } (x, y) \in R' \text{ and } x \in Q \\
\text{iff} \\
y \in R'(Q)
\end{gathered}$$

As an example, let $Q = a \vee b$ and $R = a \wedge b'$. Then

$$R(Q) \quad = \quad (\exists V.\ ((a \wedge b') \wedge (a \vee b))(V' \leftarrow V)$$

$$\begin{aligned} &= (\exists V.\ (a \wedge b'))(V' \leftarrow V) \\ &= b'(V' \leftarrow V) \\ &= b \end{aligned}$$

The inverse image $R^{-1}(Q)$ of a set Q via a binary relation R is the set of all x such that for some $y \in Q$, $R(x, y)$. If R is a QBF formula representing a relation, and Q is a QBF formula representing a set, then

$$R'^{-1}(Q) = \exists V.\ (R \wedge Q(V \leftarrow V')) \tag{2.11}$$

This can be shown by a derivation similar to the one above.

2.3.3 CTL formulas

We now have the necessary mechanics to represent Kripke structures using QBF formulas, and to characterize the CTL operators over these symbolically represented Kripke structures using QBF operators. In fact, it is only necessary to characterize the CTL operator EX, since the logical connectives have identical meanings in both logics, and the remaining CTL operators have already been characterized as fixed points of functionals using only EX and the logical operators.

To represent a Kripke structure symbolically, we will assume two sets of variables $V = \{v_1, \ldots, v_n\}$ and $V' = \{v'_1, \ldots, v'_n\}$, and a QBF formula R on $V \cup V'$ to represent the transition relation. This induces a Kripke structure $K_{V,V',R} = (S, R', L)$ where

1. The state set S is the set of truth assignments $V \to B$,
2. The transition relation R' is the relation represented by the formula R, according to (2.9),
3. The valuation L yields the truth value of each variable v_i in each state s. That is, for all $v_i \in V$, $L(s)(v_i) = s(v_i)$.

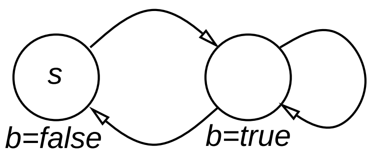
The complete procedure for symbolic model checking is characterized by the following theorem:

Theorem 4 *Let $V = \{v_1, \ldots, v_n\}$ and $V' = \{v'_1, \ldots, v'_n\}$ be disjoint sets of variables, let R be a QBF formula on $V \cup V'$, and let $K_{V,V',R}$ be* *the induced Kripke model. In this model, for all CTL formulas p and q:*

$$s \models p \quad \textit{iff} \quad s \in p, \;\; \textit{where } p \in V \tag{2.12}$$
$$s \models p \vee q \quad \textit{iff} \quad s \in (p \vee q) \tag{2.13}$$
$$s \models \neg p \quad \textit{iff} \quad s \in (\neg p) \tag{2.14}$$
$$s \models EXp \quad \textit{iff} \quad s \in (\exists V'.\ (R \wedge p(V \leftarrow V'))) \tag{2.15}$$
$$s \models E(q\ U\ p) \quad \textit{iff} \quad s \in \mu Y.\ (p \vee (q \wedge EXY)) \tag{2.16}$$
$$s \models EGp \quad \textit{iff} \quad s \in \nu Y.\ (p \wedge EXY). \tag{2.17}$$

Proof. The first three are trivial matters of definition. For (2.15), when we equate a formula with the set of states satisfying it, EXp is just $R'^{-1}(p)$, which is equal to $\exists V'.\ (R \wedge p(V \leftarrow V'))$. The last two are just theorem 2. $\square$

The above theorem shows that we can solve the model checking problem – *ie.*, determining whether a given state in a symbolically represented Kripke structure $K_{V,V',R}$ satisfies a formula f – purely by manipulations of Boolean formulas. A key point is that the Kripke structure itself is never built. Instead it is symbolically represented by a QBF formula. As an example, consider a system with one state variable b. Let the transition relation be represented by the formula $R = b \vee b'$, and let state s be $(b \leftarrow \text{false})$. The induced Kripke structure $K_{\{b\},\{b'\},R}$ is depicted below:



Let's say we want to determine whether or not $s \models EX\neg b$. According to theorem 4, we can evaluate the formula $EX\neg b$ as follows:

$$\begin{aligned} EX\neg b &= \exists b'.(R \wedge (\neg b)(b \leftarrow b')) \\ &= \exists b'.((b \vee b') \wedge (\neg b')) \\ &= \exists b'.(b \wedge \neg b') \\ &= b \end{aligned}$$

Hence, by theorem 4, $s \models EX\neg b$ iff the assignment ($b \leftarrow$ false) satisfies b, which is false.

Now consider the problem of whether or not $s \models EFb$. Using the standard fixed point algorithm, we get

$$\begin{array}{rcl} \tau^1[\text{false}] & = & b \vee EX\text{false} \\ & = & b \\ \tau^2[\text{false}] & = & b \vee EXb \\ & = & b \vee \exists b'.((b \vee b') \wedge b') \\ & = & \text{true} \\ \tau^3[\text{false}] & = & b \vee EX\text{true} \\ & = & \text{true} \end{array}$$

A fixed point is reached after two iterations. Hence, $s \models EFb$ iff the truth assignment ($b \leftarrow$ false) satisfies true, which is true.

Note that when computing least (or greatest) fixed points of τ, 2^n+1 iterations are required in the worst case, where n is the number of propositional state variables. This is the length of the longest possible strictly increasing (or decreasing) chain of subsets of S (not including the empty set), plus one extra iteration to detect the fixed point. In practice, however, the number of iterations required to reach a fixed point can be quite small.

2.3.4 Binary Decision Diagrams

It should be clear that to make the symbolic model checking technique practical, an efficient automated method for manipulating Boolean formulas is required. Fortunately, a variety of such techniques have been developed for the purpose of synthesizing digital circuits or comparing the functionality of digital circuits. These techniques may involve applying a set of rewriting rules to convert a given formula into a normal form. Alternatively, a data structure may be used to represent the formula as a Boolean function.[8] For example, a Boolean function may be

[8]Normally, when we discussing switching functions, we think of a Boolean formula as represented by a function rather than the set of satisfying truth assignments of the formula. A Boolean formula f over an ordered set of variables

represented by a truth table, or by a set of "cubes" which cover the truth table of the function, or by a binary decision tree. Each representation has associated procedures for applying Boolean operations. Any method of manipulating Boolean formulas that can implement the operations $p \wedge q$, $p \vee q$, $\neg p$, $\exists V'.p$ and $p(V \leftarrow V')$ can be used in symbolic model checking. By far the most effective method known to date, however, is the Ordered Binary Decision Diagram method developed by Bryant [Bry86].

Ordered Binary Decision Diagrams are a form of reduced decision graph that give compact canonical representation for Boolean formulas. They have been used extensively for comparison of switching functions [BBB+87, FB89]. The OBDD canonical representation for a Boolean function can be derived by reducing a related structure called an ordered decision tree. In an ordered decision tree, the value of the function is obtained by descending the tree from the root to a leaf. At each node along the path, one descends to the left child if the value of the variable labeling the node is 0, and to the right child the value is 1. Each leaf of the tree is labeled with a value 0 or 1 which gives the result of the function. The tree is said to be ordered if the variables always occur in the same order along any path from root to leaf. In this case, reading the leaves from left to right, one obtains the truth table of the function.

As an example, an ordered decision tree for the function $a \wedge b \vee c \wedge d$ is depicted in figure 2.1.

The canonical OBDD form is a directed acyclic graph which can be obtained from the ordered decision tree by the following two steps:

1. Combine any isomorphic subtrees into a single tree.

2. Eliminate any nodes whose left and right children are isomorphic.

Steps 1 and 2 can be applied in a bottom up fashion, to yield the canonical OBDD representation in linear time. Bryant called this operation *Reduce*. The size of the resulting graph is strongly dependent

$V = \{v_1, \ldots, v_n\}$ induces a function $f : \{0,1\}^n \to \{0,1\}$ in the obvious way: $f(x_1, \ldots, x_n) = 1$ iff the truth assignment $(v_i \leftarrow x_i)$ satisfies f. The two views are equivalent, but the functional representation seems to be more standard in the context of Boolean manipulation.

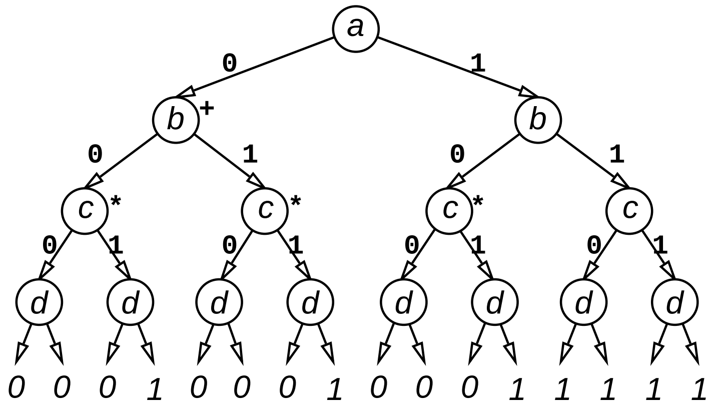


Figure 2.1: Ordered Decision Tree

on the order of the variables. This variable ordering, however, is the key to obtaining the reduced form. This is what distinguishes OBDDs from the more general class of Binary Decision Diagrams described by Akers [Ake78].

As an illustration of reduction to canonical form, consider the the ordered decision tree of figure 2.1. The three nodes marked "*" are roots of isomorphic subtrees. Thus, they can be combined into a single subtree. In addition, from the node marked "+", one arrives at the same subtree when descending to the left or right (*ie.*, independently of the value of b), hence this vertex does not affect the value of the function and may be eliminated. The result of applying the *Reduce* operation to the tree of of figure 2.1 is depicted in depicted in figure 2.2. Note the significant reduction in the number of vertices, resulting essentially from redundancy in the truth table of the function.

The canonical OBDDs are a subclass of DAGs (directed acyclic graphs) where each leaf is labeled by 0 or 1, and each non-leaf is labeled by a variable. It is most convenient to define this class inductively, by building large DAGs from smaller ones. For this reason, we will number the variables from the bottom up.[9] In the sequel, the term *dimension* will be used to denote the highest variable index occurring in a DAG.

[9]Unfortunately, this is the opposite of the numbering adopted by Bryant, but it makes the proofs clearer.

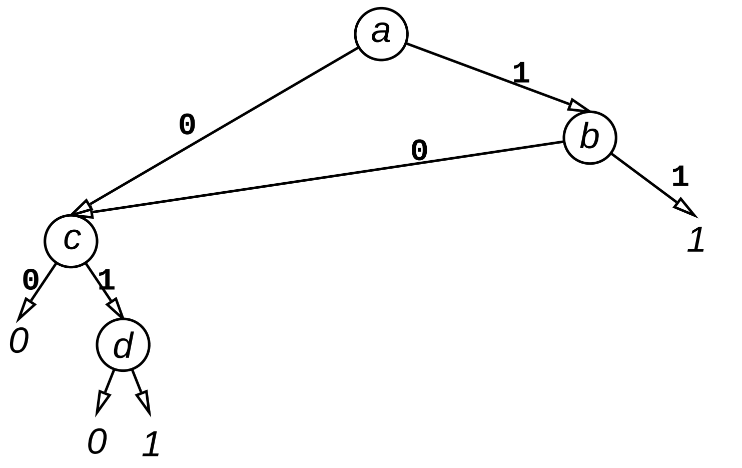


Figure 2.2: Ordered Binary Decision Diagram

We will simultaneously define the class of DAGs which are canonical OBDDs and the functions they denote, by induction on the dimension:

Definition 1 *Let V be an n-tuple $(v_1, v_2, \ldots, v_n)$ of variables. The class OBDD(V) consists of the* terminals 0 *and* 1, *and a collection of triples in $\mathcal{S} \times$ OBDD$(\mathcal{S}) \times$ OBDD$(\mathcal{S})$ called* non-terminals. *With each element p of OBDD(V), we associate a* dimension d_p, *where $0 \leq d_p \leq n$, and a Boolean function $f_p : B^n \to B$. The class OBDD$(\mathcal{S})$ is the least such that, for all $x \in \{0,1\}^n$:*

1. $0 \in$ *OBDD*(V), $d_0 = 0$, *and* $f_0(x) = 0$,
2. $1 \in$ *OBDD*(V), $d_1 = 1$, *and* $f_1(x) = 1$,
3. *if l and h are* distinct *elements of OBDD(V), where $d_l < i \leq n$ and $d_h < i$, then the triple $r = (v_i, l, h)$ is also in OBDD(V), $d_r = i$ and*
$$f_r(x) = \begin{cases} f_l(x) & \text{if } x_i = 0 \\ f_h(x) & \text{if } x_i = 1 \end{cases}$$

With regard to canonicity, the salient aspects of the above definition are that a triple (v_i, l, h) is a canonical OBDD only if l and h are *distinct* and i is greater than the dimensions of l and h (the variable ordering requirement).[10] One important consequence of this is that f_p, the function represented by a DAG p, does not depend on any variables of index greater than d_p:

Lemma 1 *For all $p \in OBDD(V)$, for all $d_p < i \leq n$, $f_p(v_i \leftarrow 0) = f_p(v_i \leftarrow 1)$.*

Proof. By induction over d_p. We assume the statement of the theorem holds for all q such that $d_q < d_p$. The terminal cases, $p = 0$ and $p = 1$ are trivial. For the non-terminal case, let $p = (v_j, l, h)$, where $j < i$. Now consider two cases, $v_j = 0$ and $v_j = 1$. In the first case, $f_p(v_i \leftarrow 0) = f_l(v_i \leftarrow 0)$ and $f_p(v_i \leftarrow 1) = f_l(v_i \leftarrow 1)$. These are equal by inductive hypothesis, since $d_l < i$. The other case, $v_j = 1$ is similar, with f_h for f_l. □

It is not difficult to show that OBDDs canonically represent the Boolean functions. That is, each Boolean function is represented by exactly one OBDD. We show first that there are no two distinct OBDDs representing the same function, and second, that every Boolean function is represented by some OBDD. The following theorem is essentially due to Bryant [Bry86], although the formalization is different, and as a result, it is hoped, the proof is substantially simpler.

Theorem 5 (Bryant) *If p and p' are elements of $OBDD(V)$, then $f_p = f_{p'}$ implies $p = p'$*

Proof. By simultaneous induction over d_p and d'_p. We assume the statement of the theorem holds for all q and q', where $d_q < d_p$ and $d_{q'} < d_{p'}$. Suppose that $f_p = f_{p'}$:

Consider first the case where $d_p = d_{p'}$. Either p and q are both terminals, (in which case $p = p' = 0$ or $p = p' = 1$) or they are both non-terminals, $p = (v_i, l, h)$ and $p' = (v_i, l', h')$. For non-terminals, we

[10]There is an alternative formulation of OBDDs due to Clarke [KC90] which does not require l and h to be distinct, but requires that $i = d_l + 1 = d_h + 1$. In this case, the OBDD for a function f is exactly the minimal DFA recognizing the language $\{x \in \{0,1\}^n \mid f(x) = 1\}$. Thinking of OBDDs as minimal DFAs can provide useful insights into the complexity of representing certain classes of functions as OBDDs.

have $f_l = f_p(v_i \leftarrow 0) = f_{p'}(v_{i'} \leftarrow 0) = f_{l'}$, and similarly $f_h = f_p(v_i \leftarrow 1) = f_{p'}(v_{i'} \leftarrow 1) = f_{h'}$. Hence, by induction, $l = l'$ and $h = h'$, so $p = p'$.

Second, consider the case where $d_p > d_{p'}$. It follows that p is a non-terminal (v_i, l, h). Further, by the previous lemma, $f_{p'}(v_i \leftarrow 0) = f_{p'}(v_i \leftarrow 1)$. Therefore, $f_p(v_i \leftarrow 0) = f_p(v_i \leftarrow 1)$, so $f_l = f_h$. By induction, then, $l = h$. This is a contradiction, however, since if l and h are not distinct, then p is not in OBDD(V).

A symmetric argument applies to the case $d_p < d_{p'}$. □

Theorem 6 *Given a function $f : B^n \to B$, there exists $p \in OBDD(V)$ such that $f_p = f$.*

Proof. By induction on i, the greatest number such that $f(v_i \leftarrow 0) \neq f(v_i \leftarrow 1)$. By inductive hypothesis, there exist q and r in OBDD(V) such that $f_q = f(v_i \leftarrow 0)$ and $f_r = f(v_i \leftarrow 1)$. Further, q and r are distinct, since $f(v_i \leftarrow 0) \neq f(v_i \leftarrow 1)$. Thus, let $p = (v_i, q, r)$. □

Because each function is represented by a unique OBDD, testing two OBDDs for functional equality can be accomplished in constant time. This property of OBDDs is useful for determining when a fixed point has been reached in the standard fixed point algorithm.

The *Apply* algorithm

Bryant describes an algorithm called *Apply*, which applies an arbitrary Boolean operation $\bullet$ to two OBDDs. The operation $\bullet$ can be any of the 16 Boolean functions of two variables – *Apply* computes the natural extension of $\bullet$ to two Boolean functions. Given two non-terminal OBDDs p and q, the *Apply* algorithm breaks the problem of computing $r = p \bullet q$ into two subproblems on the children of p and q.

Take first the case where $d_p = d_q$. Let $p = (v_i, l_p, h_p)$ and $q = (v_i, l_q, h_q)$. It is easily shown that

- $r(v_i \leftarrow 0) = p(v_i \leftarrow 0) \bullet q(v_i \leftarrow 0) = l_p \bullet l_q$ and

- $r(v_i \leftarrow 1) = p(v_i \leftarrow 1) \bullet q(v_i \leftarrow 1) = h_p \bullet h_q$.

Thus, we create two subproblems $l = l_p \bullet l_q$ and $h = h_p \bullet h_q$. On the other hand, suppose that $p = (v_i, l_p, h_p)$ and $q = (v_j, l_q, h_q)$, where $i > j$. In this case, $q(v_i \leftarrow 0) = q(v_i \leftarrow 1) = q$. So,

- $r(v_i \leftarrow 0) = p(v_i \leftarrow 0) \bullet q = l_p \bullet q$ and
- $r(v_i \leftarrow 1) = p(v_i \leftarrow 1) \bullet q = h_p \bullet q$.

Therefore, we create two subproblems $l = l_p \bullet q$ and $h = h_p \bullet q$. The remaining case, $i < j$, is symmetric.

The subproblems are solved recursively to obtain $l = r(v_i \leftarrow 0)$ and $h = r(v_i \leftarrow 1)$. From these two cofactors, we can derive r. If l and h are equal, then $r = h = l$. If they are distinct, then $r = (v_i, l, h)$. Finally, if p and q are both terminals, *Apply* simply uses the truth table for $\bullet$.

Since each subproblem of dimension d can generate two subproblems of dimension $d-1$, it might seem that this algorithm is exponential. It can be made polynomial, however, by applying dynamic programming. Notice that each subproblem is determined by a pair of OBDDs p' and q' which are descendants of p and q respectively. Hence, the maximum number of distinct subproblems is the product of the size of p and the size of q. By keeping a hash table of triples (p, q, r), we can reduce the number of recursive calls to $|p| \cdot |q|$. Bryant shows that this upper bound is tight, since there exist functions p and q for which the size of r is $|p| \cdot |q|$.

The *Compose* algorithm

Bryant also gives an algorithm called *Compose* which computes $p(v_i \leftarrow q)$, where p and q are OBDDs, and v_i is a variable. The algorithm is easily adapted for simultaneous substitution of a vector of variables. Hence, given that

$$\exists v_i . p = p(v_i \leftarrow 0) \vee p(v_i \leftarrow 1), \tag{2.18}$$

the compose procedure could be used to implement both the variable substitution operation $p(V \leftarrow V')$ and the existential quantification operation $\exists V'.p$ needed for symbolic model checking. On the other hand, a much more efficient procedure can be obtained by combining the quantification and conjunction operations in the expression for EXp into a single OBDD operation computing $\exists V'.(p \wedge q)$. Applying the quantifiers in a bottom-up fashion as the conjunction subproblems are solved results in a substantial reduction in the size of the intermediate results by reducing the number of variables.

The *AndExists* algorithm

This algorithm, which we will call *AndExists* is basically a modification of *Apply*. Let r be the OBDD representing the function $\exists V'.(p \wedge q)$. We compute r by generating subproblems l and h in the same manner as if using the *Apply* algorithm for $\bullet = \wedge$. When the results of the subproblems are obtained, if the leading variable v_i is a component of V', the result is $r = l \vee h$ (see equation 2.18). This result is obtained by calling *Apply* with $\bullet = \vee$. On the other hand, if v_i does not occur in V', then the result is the same as for *Apply*: if $l = h$, then $r = l = h$, else $r = (v_i, l, h)$.

The motivation for this algorithm is to avoid producing the entire OBDD for $p \wedge q$, which has $2n$ variables, where n is the number of state variables of the model. This is done by applying existential quantification to the results of subproblems as soon as they become available, yielding a result with only n variables. Empirically, this provides a substantial savings in space.

As in the *Apply* algorithm, a table of triples (p, q, r) is used to avoid resolving previously computed subproblems. The maximum size of this table is $|p| \cdot |q|$. However, unlike in the *Apply* algorithm, the recursive calls cannot be executed in constant time. This is because each call may require a $\vee$ operation to be performed. At present, the author is unaware of a bound on the complexity of *AndExists* better than $O(|p| \cdot |q| \cdot 2^{2n})$, which is simply the number of $\vee$ problems to be solved ($|p| \cdot |q|$ in the worst case) times the square of the largest possible OBDD size, 2^n. In practice, this number of operations has not been observed, so one might conjecture that there is a tighter bound. It seems unlikely that a polynomial bound will be found, however, since it is easily shown that if vector existential quantification on OBDDs can be computed in

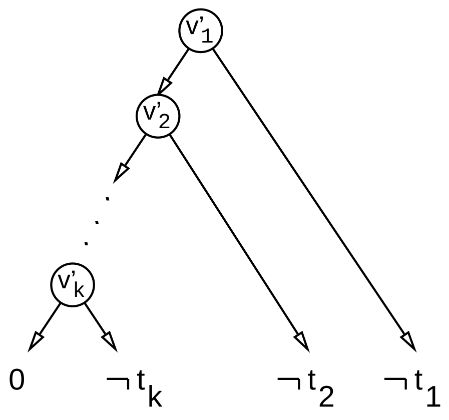


Figure 2.3: Variable ordering for 3-SAT reduction

polynomial time, then P = NP.

The proof of this is by reduction from 3-SAT, as follows: Let $f = t_1 \wedge t_2 \wedge \cdots t_k$ be a 3-SAT formula, that is, $t_i = (x_i \vee y_i \vee z_i)$, where x_i, y_i and z_i are positive or negative literals. The OBDD representation of each t_i has no more than 3 non-terminals. Now introduce new variables $V' = (v'_1, v'_2, \ldots, v'_k)$, corresponding to the terms of f, and let

$$f' = \bigvee_{1 \leq i \leq k} (\neg t_i \wedge v'_i \wedge \bigwedge_{1 \leq j < i} \neg v'_j)$$

For a suitable variable ordering, the OBDD representing f' has no more than $4k$ non-terminals (see figure 2.3), hence can be built in polynomial time. The formula f is satisfiable iff $\exists V'.f' \neq 1$. Thus, if $\exists V'.f'$ can be computed in polynomial time, then P = NP.

As an aside, it is not difficult (though a bit tedious) to show that the symbolic CTL model checking problem is PSPACE-complete. To show PSPACE-hardness, one starts with a polynomial space bounded Turing machine, introduces a sufficient number of Boolean variables to encode the entire tape, plus the pointer and the finite control, then expresses the transition relation of the entire system as a QBF formula. To show that the problem is in PSPACE, one can show that the problem can be reduced to satisfiability of a QBF formula of polynomial size, using the "iterative squaring" technique of Burch, *et al.* [BCM+90]. Details are left to the reader.

2.4 Examples

Although the worst case complexity of symbolic model checking is high (using OBDDs or other Boolean function representations), in practice the worst case complexity is rarely achieved, and the symbolic technique can in some cases be dramatically more efficient than previous methods. As an illustration of this, let's look at two hardware examples – a synchronous fair bus arbiter, and an asynchronous distributed mutual exclusion ring circuit (the one studied by David Dill in his thesis [Dil88] and designed by Alain Martin [Mar85]).

2.4.1 Synchronous state machines

For a synchronous finite state machine, the transition relation can be given as a conjunction of Boolean formulas, each determining the new state of one register as a function of its old state and the inputs. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of Boolean variables representing the state of the registers in the circuit, and let $W = \{w_1, w_2, \ldots, w_m\}$ be a set of variables representing the values of the inputs to the circuit. For all $i = 1 \ldots n$, let $f_i[V, W]$ define the value of register i in the next state, in terms of V and W. The transition relation of the state machine can be expressed as a Boolean formula in the following form:

$$R = \bigwedge_{i=1}^{n} R_i, \quad \text{where } R_i = (v_i' \iff f_i[V, W]). \tag{2.19}$$

In general, for models of synchronous systems, the transition relation is a conjunction of formulas representing the individual components of the system, since transitions of the components are *simultaneous*. The outputs of the state machine can be given as Boolean functions of the inputs and registers. These functions can be substituted for atomic propositions in CTL formulas, so there is no need to introduce variables to represent the outputs.

As an example of a synchronous state machine, we will consider a synchronous bus arbiter circuit. The purpose of the bus arbiter is to grant access on each clock cycle to a single client among a number of clients contending for the use of a bus (or other resource). The

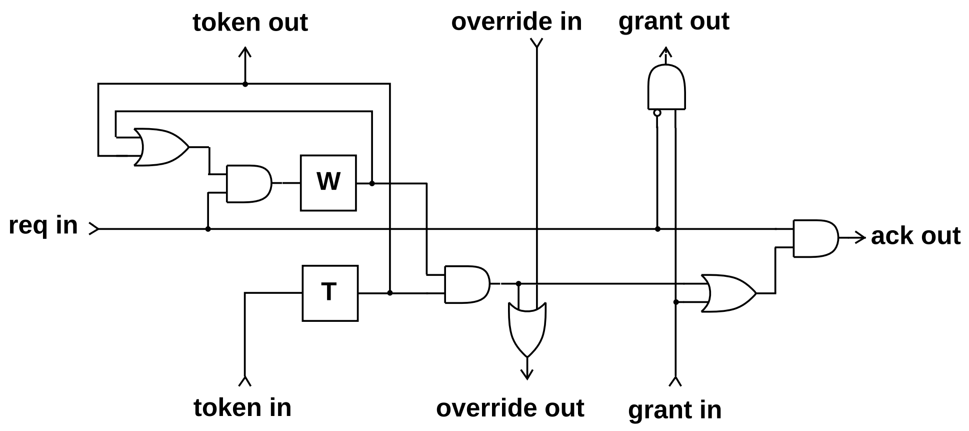


Figure 2.4: Cell of synchronous arbiter circuit

inputs to the circuit are a set of request signals $req_0 \ldots req_{k-1}$, and the outputs are a set of acknowledge signals $ack_0 \ldots ack_{k-1}$. Normally, the arbiter asserts the acknowledge signal of the requesting client with the lowest index. However, as requests become more frequent, the arbiter is designed to fall back on a round robin scheme, so that every requester is eventually acknowledged. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every clock cycle. If a given client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus.

The basic cell of the arbiter is depicted in figure 2.4.1. This cell is repeated k times, as shown in figure 2.4.1. Each cell has a request input and an acknowledge output. The grant output of cell i is passed to cell $i + 1$, and indicates that no clients of index less than or equal to i are requesting. Hence, a cell may assert its acknowledge output if its grant input is asserted. Each cell has a register T which stores a one when the token is present. The T registers form a circular shift register which shifts up one place each clock cycle. Each cell also has a register W (for "waiting") which is set to one when the request input is asserted and the token is present. The register remains set while the request persists, until the token returns. At this time, the cell's override and acknowledge outputs are asserted. The override signal propagates

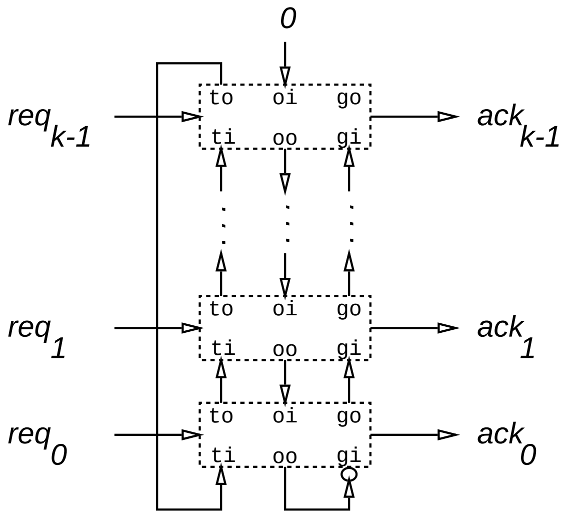


Figure 2.5: Configuration of the synchronous arbiter circuit

through the cells below, negating the grant input of cell 0, and thus preventing any other cells from acknowledging at the same time. The circuit is initialized so that all of the W registers are reset and exactly one T register is set.

The desired properties of the arbiter circuit are:

1. No two acknowledge outputs are asserted simultaneously
2. Every persistent request is eventually acknowledged
3. Acknowledge is not asserted without request

Expressed in CTL, they are:

1. $\bigwedge_{i \neq j} AG \neg(\mathrm{ack}_i \wedge \mathrm{ack}_j)$
2. $\bigwedge_i AGAF(\mathrm{req}_i \Rightarrow \mathrm{ack}_i)$
3. $\bigwedge_i AG(\mathrm{ack}_i \Rightarrow \mathrm{req}_i)$

Using the symbolic CTL model checking procedure, we can determine whether the design has these properties, for a given number of cells. Figure 2.6 plots the performance of the symbolic model checking procedure for this example in terms of several measures: the size of the transition relation in OBDD nodes, the total run time (on a Sun3, running an implementation in the C language), and the maximum number of OBDD nodes used at any given time.[11] We observe that as the number of cells in the circuit increases, the size of the transition relation increases linearly (in section 2.5, we will prove a theorem that shows why this is the case). The execution time is well fit by a quadratic curve. The number of reachable states, however, explodes exponentially (note the logarithmic scale on the reachable states axis).

To obtain polynomial performance for this example, it was necessary to add a wrinkle to the symbolic model checking algorithm. In the first experiment it was found that although most of the specification was checked quickly, the time required to check property 2 for cell 0 doubled each time a cell was added. The reason for this is rather remarkable. Consider a function called *Rotate*, which returns true for a pair of n bit binary numbers when one number can be obtained from the other by a rotation of j bits. There is no variable ordering which yields an efficient OBDD for this function for all j.[12] In fact, a very similar function occurs in computing the set of states satisfying the formula $AF(\mathrm{req}_0 \Rightarrow \mathrm{ack}_0)$, where the two binary numbers are given by the W and T registers respectively. Note that, for cell 0, request implies acknowledge exactly when no other cell has both W and T registers set. The T registers rotate once per clock cycle. Thus, $\mathrm{req}_0 \Rightarrow \mathrm{ack}_0$ is necessarily true j steps in the future exactly when there is no other cell i for which $W_i \wedge T_{i-j \bmod k}$. The OBDD representing this set of states grows exponentially in the number of cells.

This illustrates a fairly general phenomenon: circuits tend to be "well behaved" in the part of their state space which is reachable from the initial state, but not elsewhere. In the case of the synchronous arbiter, only states with one T register set are reachable. However,

[11]The latter number should be regarded as being accurate only to within a factor of two, since the garbage collector in the implementation scavenges for unreferenced nodes only when the number of nodes doubles.

[12]This can be shown using the technique of [Bry91]. It is sufficient that for any variable order there is some rotation such that when the order is cut in half, information proportional to n must be passed from one half to the other.

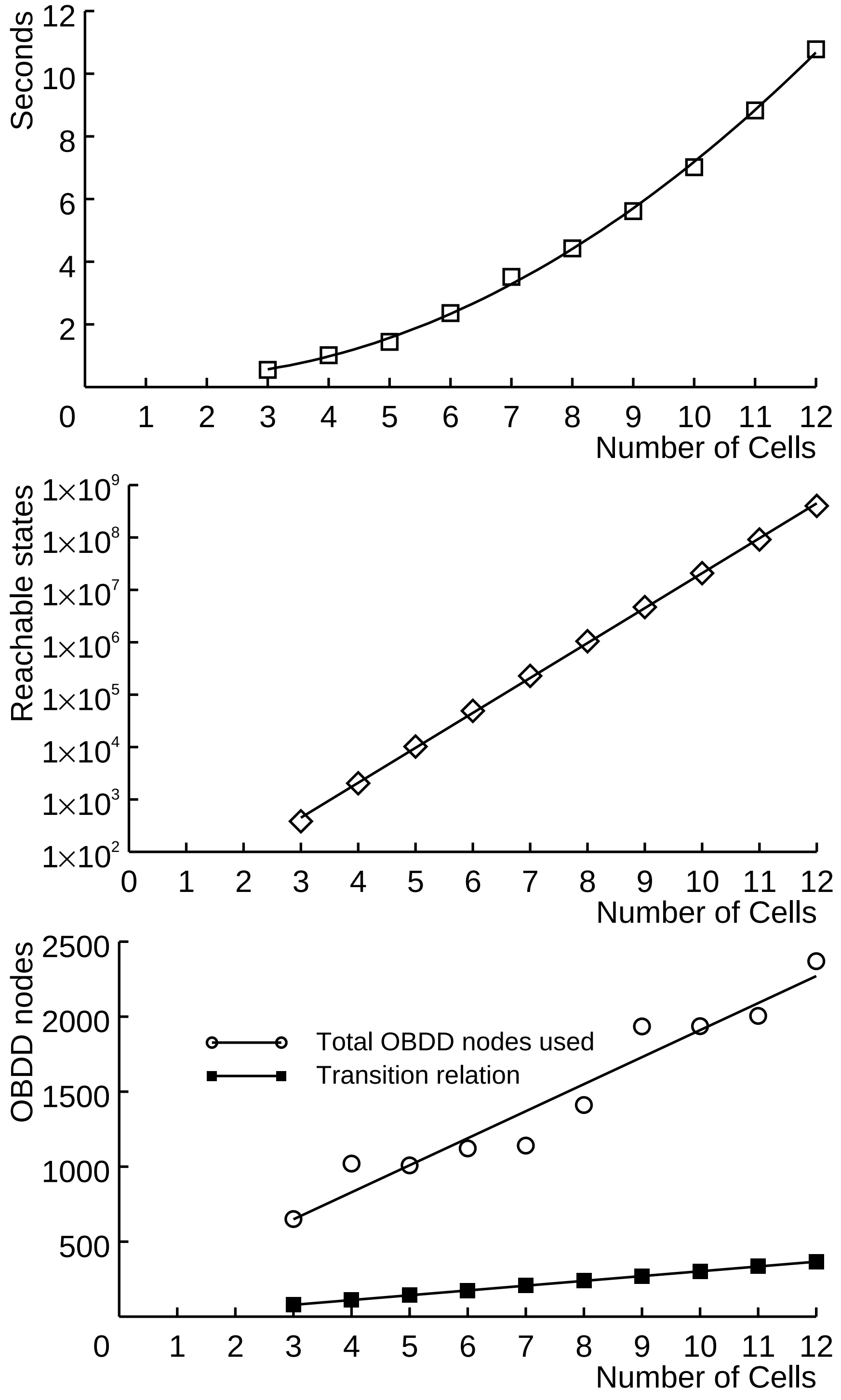


Figure 2.6: Performance – synchronous arbiter example.

the symbolic model checking technique considers all states, including states with multiple tokens. A good solution to this problem in general is first to compute the set of reachable states, and then to restrict all of computations of the CTL model checking algorithm to those states. Since the reachable states are closed under the transition relation, this has no effect on the truth value obtained for formulas at the initial state. In particular, this solves the problem of the bus arbiter circuit, since in its reachable state space, the T registers cannot represent an arbitrary binary number.

The set of reachable states is the least fixed point of

$$\tau[Y] = I \vee R(Y)$$

where I is the set of initial states. Applying the standard fixed point algorithm in this case effectively yields a forward breadth first search of the state space. By computing the reachable states first and then using this set to restrict the CTL model checking algorithm, we obtain the polynomial run time results described above. This technique is also used for other experiments described in the sequel, unless otherwise noted.

2.4.2 Asynchronous state machines

In an asynchronous state machine, there is no global clock to which all state changes are synchronized. This makes designing correct asynchronous circuits considerably more challenging than designing correct synchronous circuits. We will consider two plausible models of asynchronous state machines. In the first, which we will call the *simultaneous* model, any or all state variables may change state in a given transition. Each state component makes an independent and non-deterministic choice regarding whether to change value or not. In the second model, which we will call the *interleaving* model, only one state component changes value in a given transition. The choice of which component changes value is non-deterministic.[13] In either model, we

[13] A discussion of which state machine model is more suitable for circuit design is beyond the scope of this work. In general, conditions would have to be imposed on either model in order to make it implementable in a given design style. For discussion of asynchronous design techniques, see [MB59, Sei80b].

consider an asynchronous state machine composed of n gates. We will use state variable v_i to stand for the output of gate i, and $f_i[V, W]$ to represent the function computed by gate i (where V is the set of state variables, and W the set of inputs).

In the simultaneous model, the transition relation can be represented by a formula in the form:

$$R = \bigwedge_{1 \leq i \leq n} R_i, \quad \text{where } R_i = (v_i' \iff f_i[V, W]) \vee (v_i' \iff v_i). \qquad (2.20)$$

For any transition and any state variable v_i, either the new value of v_i is determined by $f_i[V, W]$, or it is the same as the old value. Note that this differs from the synchronous model (2.19) in which every state variable is reevaluated at every transition.

In the interleaving model, the transition relation can be represented by a formula in the form:

$$\bigvee_i R_i, \quad \text{where } R_i = (v_i' \iff f_i[V, W]) \wedge (\textstyle\bigwedge_{j \neq i}(v_j' \iff v_j)) \qquad (2.21)$$

In any transition, *for some* state variable v_i, the new value of v_i is determined by $f_i[V, W]$, and the remaining variables keep their old value. Note that in this case, the transition relation is represented by a *disjunction* of component relations rather than a conjunction.

In general, for models of parallel processes whose actions interleave arbitrarily, the transition relation is disjunctive. If this is the case, we can make an easy optimization in the symbolic model checking technique: we observe that the set of states reachable by one step of the system is the union of the sets of states reachable by one step of each individual component. This is reflected in the fact that existential quantification distributes over disjunction. Thus:

$$\begin{aligned} EXp &= \exists V'.\ ((\bigvee_i R_i) \wedge p(V \leftarrow V')) \\ &= \bigvee_i \exists V'.\ (R_i \wedge p(V \leftarrow V')) \end{aligned}$$

Using this equality, we can avoid computing the transition relation of the system and instead use only the transition relations of the indi-

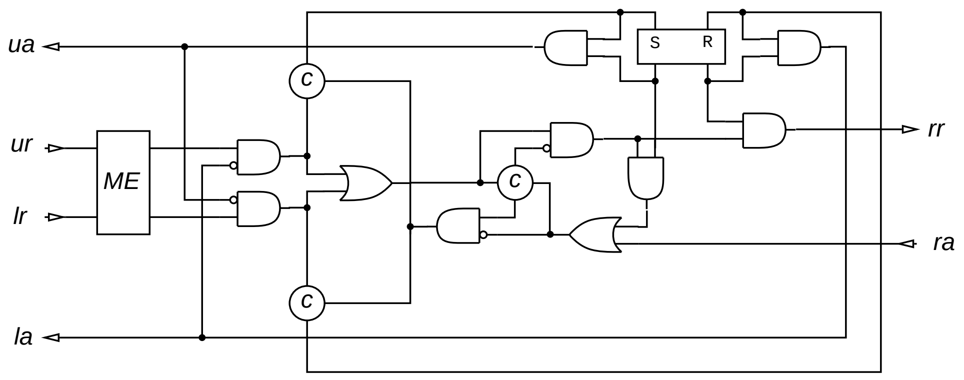


Figure 2.7: One cell of the DME circuit

vidual processes. This technique is called early quantification[14] – by rearranging the computations, we apply quantification before the logical disjunction operation. Heuristically, quantification tends to reduce OBDD size, since it reduces the number of variables. Hence, the size of the intermediate results is usually reduced (though the final result is the same).

Our example of an asynchronous state machine is the distributed mutual exclusion (DME) circuit of Alain Martin [Mar85]. It is a speed-independent circuit [Sei80b] and makes use of special two-way mutual exclusion circuits as components. Figure 2.7 is a diagram of a single cell of the distributed mutual-exclusion ring. The circuit works by passing a token around the ring, via the request and acknowledge signals *lr* and *la* on the left and *rr* and *ra* on the right. A user of the DME gains exclusive access to the resource via the request and acknowledge signals *ur* and *ua*.

The specifications of the DME circuit are as follows:

1. No two users are acknowledged simultaneously.

2. An acknowledgment is not output without a request.

3. An acknowledgment is not removed while a request persists.

[14]The *AndExists* algorithm of section 2.3.4, which combines conjunction and quantification in a bottom-up manner is also an example of early quantification.

4. All requests are eventually acknowledged.

We will consider only the first specification, regarding mutual exclusion. The others are easily formulated in CTL, although the last requires the use of fairness constraints (see section 2.6.1) to guarantee that all gate delays are finite. The formalization of the mutual exclusion specification is

$$\bigwedge_{i \neq j} AG\neg(ua_i \wedge ua_j)$$

Now let's look at the performance of the symbolic model checking algorithm in checking this formula, for both a simultaneous and an interleaving model of the circuit. For the interleaving model, we use the early quantification technique. Figure 2.8 plots the relative performance for the simultaneous model (method 1) and the interleaving model (method 2). Part (a) shows the run time as a function of the number of DME cells, part (b) shows the total storage used (measured in OBDD nodes) and part (c) shows the number of nodes used to represent the transition relation. For the moment, disregard the curves for method 3. The experiment was run for up to 7 cells of the simultaneous model (limited by space) and up to 10 cells of the interleaving model (limited by time). Part (b) of the figure shows the substantial space advantage of the interleaving model, and from part (c), we can see that most of the difference is accounted for by the savings in representing the transition relation using early quantification. In both cases, the space used is linear in the number of cells. However, we note that the increase in run time appears to be cubic for the simultaneous model, but quartic for the interleaving model. It would seem that if enough storage were available to continue the curve for method 1, the two curves would meet in the neighborhood of 10 cells.

The different asymptotic performance for the simultaneous and interleaving models can be understood by looking at the OBDDs that occur in the fixed point iterations computing the reachable states. Figure 2.9 plots the size of the largest such OBDD for each method. We can see clearly that the size is increasing linearly for the simultaneous model, but quadratically for the interleaving model. This is a phenomenon which occurs generally when comparing simultaneous *vs* interleaving models. It can be understood by considering a very simple system composed of n processes, each with states 0 and 1, and each alternating non-deterministically between these two states. If we start the system with all processes in state 0, what do we observe after k steps? In the simultaneous case, after one step, all possible states are reachable. In the interleaving case, however, after k steps, all global states with at most k 1's are reachable. This is a symmetric function. As Bryant noted [Bry86], all symmetric functions can be represented by a quadratic size OBDDs. The symmetry results from the fact that in an interleaving model, exactly one state component changes in a given transition, and the choice is arbitrary. In general, after k steps of such a model, the number of steps taken by each state component sums to k. Hence, in the set of states reachable after k steps, there is an induced correlation between the states of otherwise independent processes.

The simultaneous model appears to be inferior to the interleaving model from a symbolic model checking point of view, owing to the large amount of space required to represent the transition relation. Most of this, however, can be attributed to a phenomenon we observed in the previous example: systems tend to be well behaved only in their reachable state space. In the symbolic model checking technique, we represent the transition relation over the entire state space. Although representing only the reachable transitions might be more efficient, we seem to be caught in Catch 22: we need to represent the transition relation to compute the set of reachable states. We can avoid this problem by incrementally computing only as much of the transition relation as is necessary to compute the next iteration of the fixed point algorithm. Recall that the reachable state set is the least fixed point of $\tau[Y] = I \vee R(Y)$. By rearranging the fixed point computation slightly, we only need represent R correctly for those transitions (x, y), where x is on the "frontier" of the search:

$$\begin{aligned}\tau^{i+1}(\text{false}) &= \tau^{i}(\text{false}) \vee R(\tau^{i}(\text{false}))\\ &= \tau^{i}(\text{false}) \vee R(\tau^{i}(\text{false}) - \tau^{i-1}(\text{false}))\end{aligned}$$

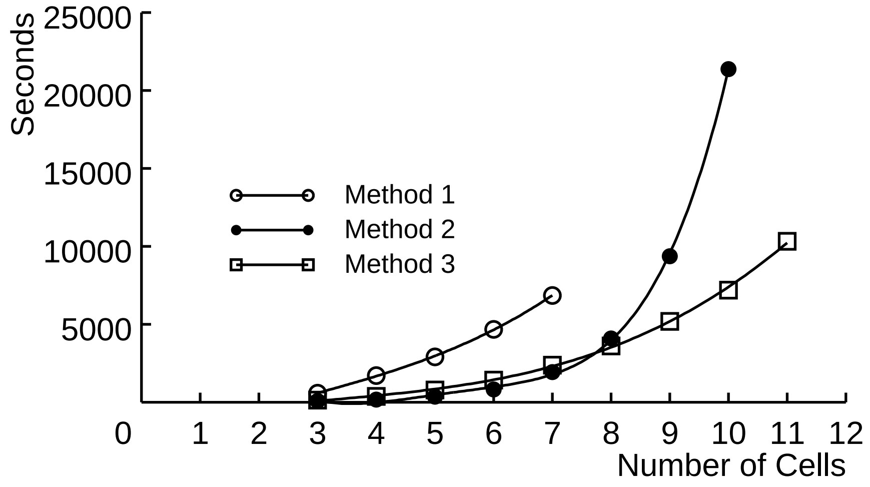
At each iteration, we can reevaluate the formula R over the set of states $\tau^{i}(\text{false}) - \tau^{i-1}(\text{false})$. This can be done by restricting each subformula using either the logical *and* or using the *Restrict* operator of Coudert, Madre and Berthet (see section 2.8). This results in a sequence of approximations to the transition relation which are substantially more compact than the complete transition relation, although we must reevaluate R at each iteration, rather that evaluating it once at the beginning. We will call this method 3.

In part (a) of figure 2.8, we see that the time used by this method, while still cubic, is a substantial improvement over the previous method for the simultaneous model (method 1). More importantly, the space used is dramatically improved, allowing a model with a larger number of cells to be checked. The method overtakes the interleaving model in run time at about 8 cells, owing to its better asymptotic performance.

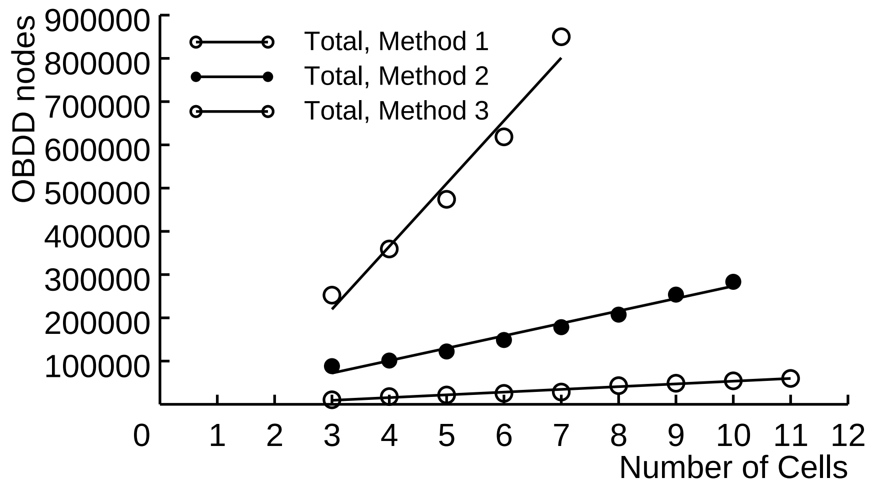
Figure 2.10 plots the number reachable states as a function of the number of cells (the numbers are indistinguishable for the two models). The number of reachable states grows exponentially in the number of cells, though not as rapidly as the total number of states, which is 2^{18n}. The key point is that for all three methods, the space and time necessary for the symbolic model checking method is polynomial in the number of cells. Thus, the state explosion problem has been avoided. The overall time complexity of $O(n^3)$ for the simultaneous model derives from three factors: a linear increase in the transition relation OBDD, a linear increase in the state set OBDDs obtained in the fixed point iterations, and a linear increase in the number of iterations. For the interleaving model, the quadratic increase in the state set OBDDs results in an overall $O(n^4)$ time complexity. On the other hand, the number of reachable states increases roughly a factor of ten with each added cell.

It is not immediately clear that either the interleaving or simultaneous model is preferable in general. Interleaving models seem to be better when the number of asynchronous processes is small, and simultaneous when the number is large. The cache consistency protocol of chapter 4 is an example of a large system with a fairly small number of complex asynchronous processes. This is an appropriate application of an interleaving model.

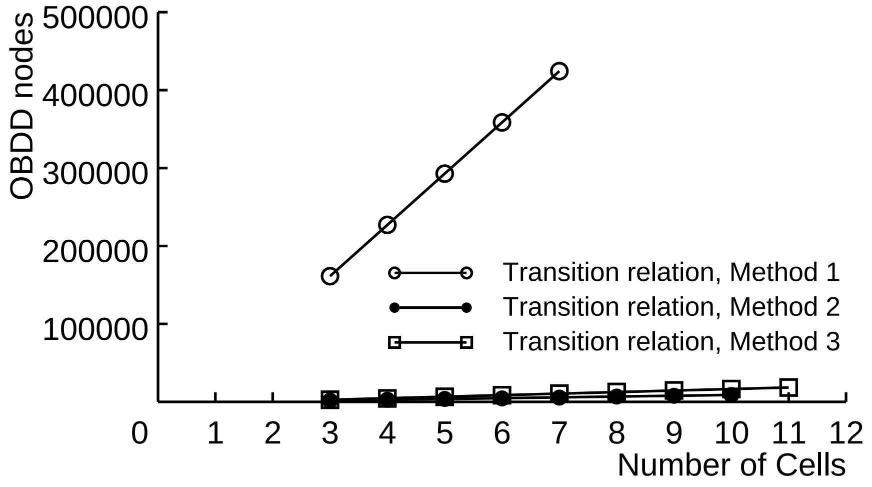
The polynomial performance of the symbolic model checking algorithm, in spite of the exponential increase in states, makes it possible to analyze fairly large instantiations of the two example circuits (the synchronous arbiter and the DME circuit). It should be possible to verify these and similar circuits for any reasonable fixed number of cells. This



(a)



(b)



(c)

Figure 2.8: Performance for DME circuit example

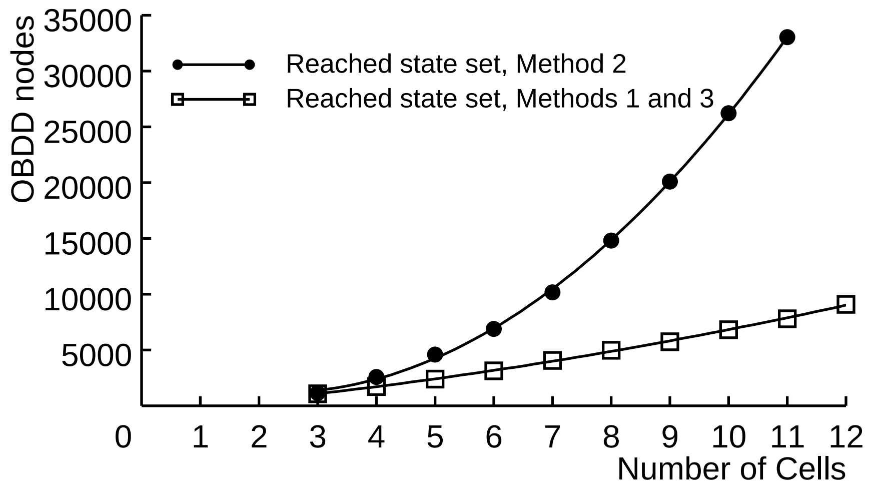


Figure 2.9: State set size for DME circuit example

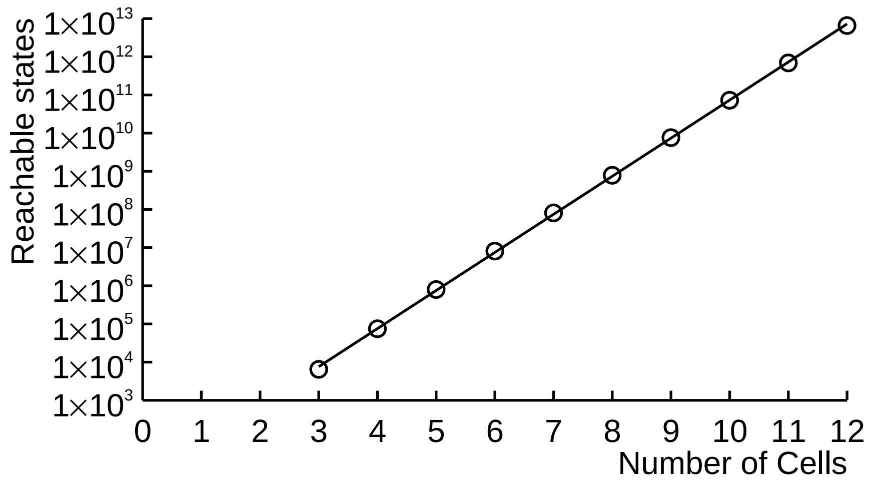


Figure 2.10: Reachable states for DME circuit example

begs the question – how many cells do we need to analyze to be guaranteed that the design is correct for any number of cells? Intuitively, for sufficiently large n, a sequence of $n + 1$ cells should be equivalent in some sense to a sequence of n cells. But in what sense equivalent? This problem is dealt with in chapter 5, where we consider induction over processes.

2.5 Graph width and OBDDs

In this section, we consider the asymptotic growth of OBDDs representing certain topological classes of circuits. This analysis explains some of the performance results of the previous section.

In 1989, Berman proved a bound on the OBDD size needed to represent circuits of bounded width. A circuit has bounded width if its elements can be arranged in a linear order such that any cut through the order crosses at most a bounded number of wires w, called the width of the circuit. There exists a variable ordering such that the OBDD size is bounded by $n2^w$, where n is the number of primary inputs of the circuit. This result applies only if the order is "topological", meaning essentially that the direction of all the wires follows the ordering. Here, this result is generalized, to show that if w_f bounds the number wires through any cut in the forward direction, and w_r bounds the number in the reverse direction, then the OBDD size is bounded by $n2^{w_f 2^{w_r}}$. For the case where $w_r = 0$, this is the same as Berman's result. Using this result, we can linearly bound the OBDD representation for the transition relation of circuits like the arbiter and the DME ring, which have linear arrangements with a bounded number of wires through any crosss section.

Fujita states that tree circuits using only AND, OR and XOR gates have linearly bounded OBDD representations [FMK90]. Here, we show that a more general class of circuits with bounded "tree width" and arbitrary function elements have polynomially bounded OBDDs. The essence of the argument is to show that these circuits can be arranged in a linear order with a width that is logarithmic in the number of gates. This yields a bound on the OBDD size which is polynomial in the number of gates.

2.5.1 Bounded width circuits

Let $L = (G, <)$ be a linear order on the gates of a circuit. We classify the primary inputs and outputs of the circuit as special instances of gates in order to simplify the definitions, and assume that the primary output is at the top of the order. Given an order L, we will say that the *forward cross section* of the circuit at gate g is the set of wires connected to an output of some gate g_1 and an input of some gate g_2 such that $g_1 \leq g$ and $g < g_2$. The *reverse cross section* is the set of wires connected to an output of some gate g_1 and an input of some gate g_2 such that $g_2 \leq g$ and $g < g_1$. We assume that no wire is connected to the outputs of two distinct gates, so these two sets are disjoint. We also assume that there are no cycles in the circuit, to insure that the circuit computes a function. The order L is said to be *topological* when all of the reverse cross sections are empty.

The forward width of the circuit under order L, denoted w_f, is the maximum size of the forward cross section at any gate g. Similarly, the reverse width of the circuit under order L, denoted w_r is the maximum size of the reverse cross section at any gate g.

The cross section of an OBDD at level i is the set of nodes labeled with variable v_i. Note that in this section, we will number the variables of the OBDD from the top down, since this makes the proofs simpler. The width w_p of an OBDD p is the maximum size of any cross section of p. The size of an OBDD is the sum of the sizes of its cross sections. Thus, the OBDD size if bounded by $n \cdot w_p$, where n is the number of variables.

It is easily shown that the size of the cross section of an OBDD at level i is the number of distinct functions

$$f_{px}(v_i, \ldots, v_n) = f_p(x_1, \ldots, x_{i-1}, v_i, \ldots, v_n)$$

which depend on v_i, where $x = (x_1, \ldots, x_{i-1})$ is a Boolean vector and f_p is the function represented by p. This observation leads to the following theorem bounding the size of an OBDD in terms of the forward and reverse widths of the circuit it represents:

Theorem 7 *If a circuit computing function f has forward width w_f and reverse width w_r for some linear order L, then there is an OBDD*

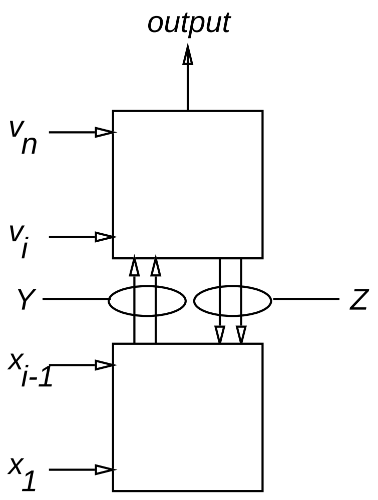


Figure 2.11: Proof of bounded width theorem

p representing function f of size bounded by $n2^{w_f 2^{w_r}}$*, where n is the number of inputs of the circuit.*

Proof. Associate the variables $v_1, \ldots, v_n$ of the OBDD with the inputs of the circuit, such that for all $i \leq j$, $v_i \leq v_j$. We can bound the size of the ith cross section of the resulting OBDD as follows. Let $x = (x_1, \ldots, x_{i-1})$ be a Boolean vector. Split the circuit in half by choosing any gate g such that $v_{i-1} \leq g < v_i$, letting Y be the forward cross section at g and Z the reverse cross section. This situation is depicted in figure 2.11. For any given value of x, Y is a function of Z, and this function determines $f_x(v_i, \ldots, v_n)$. The number of Boolean functions with $|Z|$ inputs and $|Y|$ outputs is $2^{|Y|2^{|Z|}}$ (to see this, count the number of entries in the truth table). This bounds the total number of distinct functions f_x, which in turn bounds the width of the OBDD representing f at level i. We know that $|Y| \leq w_f$ and $|Z| \leq w_r$. Thus, the overall OBDD size is bounded by $n \cdot 2^{w_f 2^{w_r}}$. □

This bound is linear in the number of inputs, exponential in the forward width and doubly exponential in the reverse width. The double exponential appears to be necessary. This can be shown using the "hidden weighted bit" function of [Bry91] as a counterexample. This circuit can be ordered in such a way that between any two inputs there is a cross section with $O(\log_2 n)$ wires in each direction, yet there is an exponential lower bound on its OBDD size. If we could bound the OBDD size with a single exponential in both the forward and reverse widths, the OBDD size would be $O(n2^{k \log_2 n}) = O(n^{k+1})$ where k is a constant.

The theorem is concerned with a single output of a combinational circuit, but it can also be applied to the transition relation of a sequential circuit. To do this, we simply transform the sequential circuit into a combinational circuit which computes the transition relation of the sequential circuit. This is done by adding a pair of inputs v_i and v_i' to represent the old and new values of each state component. Since the transition relation of the circuit is the conjunction of the transition relations of its components, we can do this while increasing the width of the circuit by only one wire in the forward direction as depicted in figure 2.12. Thus, for bounded width sequential circuits (even with wires in both directions), the size of the OBDD representing the transition relation is linear in $n_i + n_s$, where n_i is the number of inputs and n_s is the number of state components. The synchronous arbiter circuit and the DME circuit of the previous section provide experimental confirmation of this.

We have shown for a certain structural class of circuits that the representation of the transition relation is linearly bounded in the size of the circuit. We should note that in the symbolic model checking algorithm, we also use OBDDs to represent the set of states labeled with a given CTL formula. Unfortunately, we cannot expect to polynomially bound the size of the OBDDs representing these sets based purely on structural considerations. The simplest example of this is probably a circuit that inputs a binary number, stores one copy of it, then serially rotates the original by an arbitrary number of bits. This circuit has the simplest structure we might hope for that has any communication at all between the components, yet there is no variable order which yields a compact OBDD for the reachable state set of this circuit, since it implements the rotate function. The same argument would apply to a serial multiplier circuit. In general, if a circuit computes a function serially which cannot be represented by a compact OBDD, then we

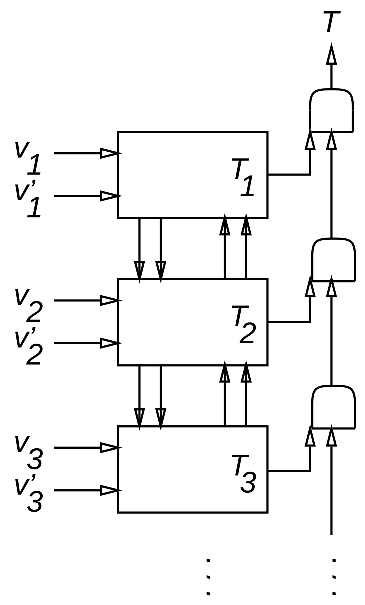


Figure 2.12: Computing a conjunctive transition relation

cannot expect the symbolic model checking algorithm using OBDDs to be efficient.

2.5.2 Bounded tree-width circuits

In the previous section, we considered the OBBD representation of circuits whose gates can be arranged in a sequence with a bounded number of wires in each cross section. Now we consider the slightly more general class of circuits which can be can be arranged in a tree with a bounded width property. This is not to say that the topology of the circuit must be a tree; rather, it must be possible to lay a spanning tree over the circuit in such a way that the width of the circuit across any arc of the spanning tree is bounded. This notion of bounded tree-width is defined as follows.

Let $T = (G, <)$ be a tree order over the gates of a circuit, where $g' < g$ iff g' is a descendant of g. Let b be the branching degree of T (*ie.*, the maximum number of children of any gate). As before, the forward cross section at node g is the set of wires connecting an output of g_1

and an input of g_2 such that $g_1 \leq g$ and $g < g_2$. Similarly, the reverse cross section of T at node g is the set of wires connecting an output of g_1 and an input of g_2 such that $g_2 \leq g$ and $g < g_1$. The forward width of the tree w_f is the size of the largest forward cross section, while the reverse width w_r is the size of the largest reverse cross section.

For the moment, let us consider the case $w_r = 0$, and let the width w stand for the forward width:

Lemma 2 *For any topological tree order $T = (G, <)$, with width w and branching degree $b > 1$, there is a topological linear order $L = (G, <')$, with width $w' \leq w(b-1)log_2|G|$.*

Proof. By induction over $|G|$, the number of gates. The base case, $|G| = 1$, is trivial. Assume the theorem holds for all circuits of size less than $|G|$. Let g be the root of the tree, and let $G_1, \ldots, G_k$ be the subtrees of the root, where $k \leq b$, and $|G_1| \leq \cdots \leq |G_k|$. By inductive hypothesis, there exist linear orders $L_i = (G_i, <_i)$ of width $w_i \leq w(b-1)log_2|G_i|$, for all $1 \leq i \leq k$. Let $L = (G, <')$ be the extension of these orders such that $G_k <' \cdots <' G_1 <' g$, as depicted in figure 2.13. The width w' of L is bounded by $\max_{1 \leq i \leq k}(w_i + (k-i)w)$. Therefore, for some i,

$$w' \quad \leq \quad w_i + (k-i)w$$

In the case $k = i$, we have

$$w' \leq w_k \leq w(b-1)log_2|G_k| \leq w(b-1)log_2|G|$$

Otherwise, $i < k$ and

$$\begin{aligned} w' &\leq w(k-i+(b-1)\log_2|G_i|) \\ &\leq w(b-1)\log_2\left(2^{\frac{k-i}{b-1}}|G_i|\right) \end{aligned}$$

Here, we note that $|G_i| \leq (\Sigma_{i \leq j \leq k}|G_j|)/(k-i+1) \leq |G|/(k-i+1)$, so

$$w' \quad \leq \quad w(b-1)\log_2\left(\frac{2^{\frac{k-i}{b-1}}}{k-i+1}|G|\right)$$

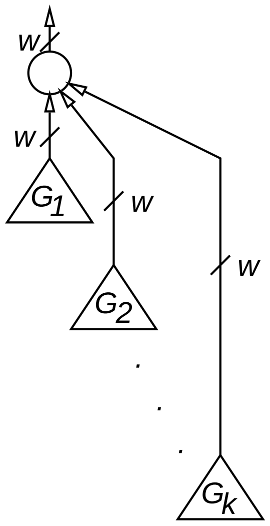


Figure 2.13: Arrangement of bounded width tree

We note that since $i \geq 1$ and $k \leq b$, $k - i \leq b - 1$. Therefore $2^{\frac{k-i}{b-1}} \leq 2$. Further, since $i < k$, $k - i + 1 \geq 2$. Thus $\frac{2^{\frac{k-i}{b-1}}}{k-i+1} \leq 1$. Therefore,

$$w' \;\leq\; w(b-1)\log_2 |G|$$

□

The theorem says that from any topological tree order of width w we can derive a linear order of width $w' \leq w(b-1)\log_2 |G|$. It follows by the previous theorem that the OBDD size is bounded by $n2^{w'} \leq n2^{w(b-1)\log_2 |G|} = n|G|^{w(b-1)}$, where n is the number of primary inputs. This bound is polynomial in the size of the circuit for a fixed width and branching factor.

Now we turn to the question of tree orders that are not topological (*ie.*, bounded tree-width circuits with both forward and reverse wires). In this case, a logarithmic bound on the width of the linear order L is not sufficient, because the OBDD size can be doubly exponential in the number of *reverse* wires.

We can still obtain a polynomial bound in n, however, by converting a tree ordered circuit with reverse wires into a functionally equivalent tree ordered circuit with only forward wires:

Lemma 3 *If $T = (G, <)$ is a tree order over a circuit computing function f, with forward width w_f and reverse width w_r, then there is a circuit computing f with* topological *tree order $T' = (G', <')$ of forward width $w'_f \leq w_f 2^{w_r}$.*

Proof. Consider H, a subtree rooted at gate h, letting Y be the forward cross section at h, and Z the reverse cross section at h. Let $h_1, \ldots, h_k$ be the children of h, and let $Y_1, \ldots, Y_k$ and $Z_1, \ldots, Z_k$ be their respective forward and reverse cross sections. This situation is depicted in figure 2.14. Let the output functions computed by H be

$$Y = f(Z, Y_1, \ldots, Y_k)$$

and for $1 \leq i \leq k$, let

$$\begin{aligned} Z_i &= r_i(Z, Y_1, \ldots, Y_k) \\ Y_i &= f_i(Z_i) \end{aligned}$$

We show by induction over $|H|$ that there exists a tree circuit H' of forward width $w'_f \leq w_f 2^{w_r}$ and reverse width $w'_r = 0$, computing the functions

$$f_x = f(x, Y_1, \ldots, Y_k), \text{ for } x \in \{0,1\}^{|Z|}$$

Note that f_x is simply row x in the truth table for Y. Since there are $2^{|Z|}$ possible values of x, and f_x has $|Y|$ components, the number of outputs of H' is $|Y|2^{|Z|}$.

By inductive hypothesis, there exist circuits H'_i for $1 \leq i \leq k$, satisfying the width bound and computing the functions

$$f_{ix} = f_i(x), \text{ for } x \in \{0,1\}^{|Z_i|}$$

Now, let h' be a gate computing f_x according to the following system of equations:

$$\begin{aligned} f_x &= f(x, f_{1x_1}, \ldots, f_{kx_k}) \\ x_i &= r_i(x, f_{1x_1}, \ldots, f_{kx_k}), \text{ for } 1 \leq i \leq k \end{aligned}$$

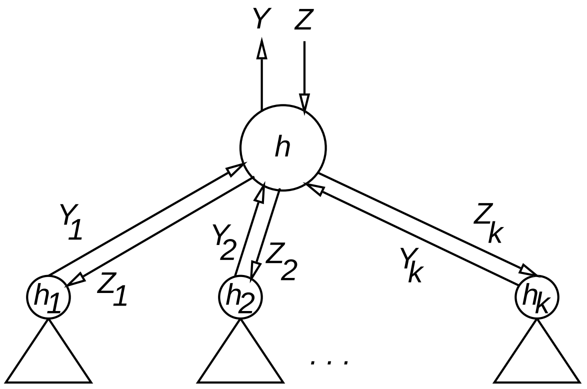


Figure 2.14: A non-topological tree order

Let H' be the tree ordered circuit obtained by taking h' as the root, and $H'_1, \ldots, H'_k$ as the children of the root. The reverse width at the root is 0, since f_{ix} does not depend on Z, and the forward width at the root is $|Y|2^{|Z|}$. Hence, using the inductive hypothesis, $w'_r = 0$ and $w'_f \leq w_f 2^{w_r}$. If h is the root node of G, then H' computes the same function as G. $\square$

This gives us the following theorem, bounding the OBDD size for tree ordered circuits with both forward and reverse wires:

Theorem 8 *If a circuit G computing function f has forward width w_f and reverse width w_r for some tree order T of branching degree $b > 1$, then there is an OBDD representing function f of size bounded by $n|G|^{w_f 2^{w_r}(b-1)}$, where n is the number of primary inputs of the circuit.*

Proof. According to lemma 3, for any tree ordered circuit of forward width w_f and reverse width w_r, we can construct a topological tree ordered circuit of width $w \leq w_f 2^{w_r}$, which computes the same function. By lemma 2, this circuit has a topological linear order L of width at most $w' \leq w(b-1)log_2|G|$. By theorem 7, there is an OBDD for the circuit of size bounded by

$$\begin{aligned} n2^{w'} &\leq n2^{w(b-1)\log_2|G|} \\ &\leq n2^{w_f 2^{w_r}(b-1)\log_2|G|} \end{aligned}$$

$$= \quad n|G|^{w_f 2^{w_r}(b-1)}$$

□

Hence, in the case of bounded tree width circuits (of a fixed branching degree), we also find that the OBDD size can be bounded polynomially in the size of the circuit. In this case, the exponent of n is related to both the width and the branching factor. Clearly, for this bound to be of any practical interest, w_f must be small, and w_r must be very small. Nonetheless, the theorem demonstrates a more general topological class of circuits with asymptotically compact OBDDs than was previously known.

2.6 Mu-Calculus model checking

The Mu-Calculus [Par74] is a logic based on extremal fixed points that is strictly more expressive than CTL,[15] and can also express a variety of properties of transition systems, such as reachable state sets, state equivalence relations, and language containment between automata. A symbolic model checking algorithm for this logic allows all of these properties to be computed using OBDDs [BCM+90].

The Mu-Calculus augments the ordinary predicate calculus in two ways. First, it allows terms to stand for relations. If f is a formula in which variables x and y are free, then f characterizes a relation – the set of all pairs (x, y) satisfying f. This relation is denoted in the Mu-Calculus by the term $\lambda x, y.f$. Second, the Mu-Calculus allows us to express least and greatest fixed points. If τ is a term, and Y is a relational (predicate) symbol, then τ is said to be formally monotonic in Y if Y always occurs under an even number of negations in τ. In this case, τ has least and greatest fixed points with respect to Y, which are denoted $\mu Y.\ \tau$ and $\nu Y.\ \tau$. A fixed point of τ with respect to Y is a relation which yields itself when substituted for all free occurrences of Y in τ.

[15]Emerson and Lei [EL86] gave a model checking algorithm for a somewhat different version of the Mu-Calculus, and showed that there are formulas in this logic that cannot be expressed in CTL. Here, we use the relational Mu-Calculus of Park [Par74]

A structure in the Mu-Calculus consists of a set $\mathcal{D}$ (the domain), a valuation ϕ for the individual symbols $\{a, b, c, \ldots\}$ and a valuation ψ for the relational symbols $\{A, B, C, \ldots\}$. The valuations assign an element from the domain to each individual symbol, and a set of n-tuples from the domain to each relational symbol. The meaning of an n-ary term τ is a set of n-tuples which we will denote $\tau[\phi, \psi]$. The 0-ary terms will be called simply propositions, and denote truth values.

The terms of the Mu-Calculus are the least set such that:

1. Every relational symbol is a term.
2. If τ is an n-ary term and $(v_1, \ldots, v_n)$ are individual symbols, then $\tau(v_1, \ldots, v_n)$ is a proposition.
3. If τ_1 and τ_2 are n-ary terms, then so are $\neg\tau_1$ and $(\tau_1 \vee \tau_2)$.
4. If p is a proposition, and v is an individual symbol, then $\exists v.\ p$ is a proposition.
5. If p is a proposition and $(v_1, \ldots, v_n)$ are individual symbols, then $\lambda v_1, \ldots, v_n.\ p$ is an n-ary term.
6. If τ is an n-ary term and Y is an n-ary relational symbol, where τ is formally monotonic in Y, then $\mu Y.\ \tau$ and $\nu Y.\ \tau$ are n-ary terms.
7. The usual abreviations are used for $\wedge$, $\Rightarrow$, $\forall$, *etc.*

The semantics of Mu-Calculus terms are defined as follows:

1. $R[\phi, \psi] = \psi(R)$, where R is a relational symbol.
2. $\tau(v_1, \ldots, v_n)[\phi, \psi]$ is true iff $(\phi(v_1), \ldots, \phi(v_n))$ is in $\tau[\phi, \psi]$.
3. $(\neg\tau)[\phi, \psi] = \mathcal{D}^n - \tau[\phi, \psi]$, $(\tau_1 \vee \tau_2)[\phi, \psi] = \tau_1[\phi, \psi] \cup \tau_2[\phi, \psi]$.
4. $(\exists v.\ p)[\phi, \psi]$ is true if for some $x \in \mathcal{D}$, $p[\phi(v \leftarrow x), \psi]$ is true.
5. $(\lambda v_1, \ldots, v_n.\ p)[\phi, \psi]$ is the set of n-tuples $(x_1, \ldots, x_n) \in \mathcal{D}^n$ such that $p[\phi(v_i \leftarrow x_i), \psi]$ is true.
6. $(\mu Y.\ \tau)[\phi, \psi]$, where τ is an n-ary term, is the least set $S \subseteq \mathcal{D}^n$ such that $S = \tau[\phi, \psi(Y \leftarrow S)]$. $(\nu Y.\ \tau)[\phi, \psi]$ is the greatest such S.

2.6.1 Applications of the Mu-Calculus

The Mu-Calculus is quite expressive, as can be seen by the following compendium of applications. To begin with, given a binary relation R, the image of a set $Q \subseteq \mathcal{D}$ *via* R is

$$R(Q) = \lambda y.\ \exists x.\ (R(x,y) \wedge Q(x))$$

The set reachable from Q in any number of steps of R (including 0) is

$$R^*(Q) = \mu Y.\ (Q \vee R(Y))$$

The transitive (irreflexive) closure of the relation R is

$$R^+ = \mu Y.\ [R \vee \lambda x, z.\ \exists y.\ (Y(x,y) \wedge Y(y,z))]$$

CTL and fairness constraints

The interpretation of the operators of CTL in a Kripke model $(\mathcal{D}, R, L)$ can be characterized in the Mu-Calculus as follows:

$$\begin{array}{rcl} EXp & = & \lambda x.\ \exists y.\ (R(x,y) \wedge p(y)) \\ EFp & = & \mu Y.\ (p \vee EXY) \\ EGp & = & \nu Y.\ (p \wedge EXY) \\ E(q\ U\ p) & = & \mu Y.\ (p \vee (q \wedge EXY)) \end{array}$$

In addition to these standard operators, we can also characterize the CTL operators under *fairness constraints.* A fairness constraint in its simplest form is a condition that is assumed to hold infinitely often along all computation paths. Such conditions can be used to enforce fair scheduling of processes and access to resources. They are not directly expressible in CTL, since the tense operators F and G cannot be directly combined. Instead, we restrict the path quantifiers of CTL to apply only to those paths along which each formula in a set C holds infinitely often. To distinguish these constrained path quantifiers from ordinary path quantifiers, we subscript them with C. Thus, $A_C f$, where C is a set of CTL formulas and f is a linear formula, means that for all paths, if each formula of C is true infinitely often, then f is true.

Similarly, the formula $E_C f$ means that there exists a path such that each formula of C is true infinitely often and f is true. Here, we consider only the CTL operators with existential path quantifiers, since the operators with universal quantifiers can be derived from these.

The formula $E_C Gp$ is true when there is some path in which p is true in every state, and each element of C is true infinitely often. Let

$$\tau[Y] = p \wedge EX \bigwedge_{c \in C} E(Y \ U \ (Y \wedge c)).$$

We argue as follows that $E_C Gp$ is the greatest fixed point of τ. First, if Y is a fixed point, then every state in Y satisfies p, and further, has a nontrivial path remaining in Y which leads to a state satisfying each fairness constraint. Hence, a looping path can be constructed satisfying each infinitely often without exiting Y. Thus $Y \subseteq E_C Gp$. On the other hand, suppose $Y = E_C Gp$. Since every state in Y has a path touching each fairness constraint infinitely, as does each state along that path, it follows that every state in Y can reach every fairness constraint without exiting Y. Thus $Y \subseteq \tau[Y]$. Therefore, $E_C Gp$ is the greatest fixed point of τ. The set of states satisfying $E_C Gp$ is expressed in the Mu-Calculus as

$$\nu Y.\ (p \wedge EX \bigwedge_{c \in C} E(Y \ U \ (Y \wedge c)))$$

The remaining operators under fairness constraints can be characterized in terms of $E_C Gp$, as follows:

$$\begin{aligned} E_C Xp &= EX(p \wedge E_C G \text{ true}) \\ E_C Fp &= EF(p \wedge E_C G \text{ true}) \\ E_C(q \ U \ p) &= E(q \ U \ (p \wedge E_C G \text{ true})) \end{aligned}$$

Emerson and Lei [EL86] give a characterization in the Mu-Calculus of CTL under a more general class of fairness constraints. Each constraint in this scheme requires that one condition holds infinitely often or a second condition holds finitely often (for example, either acknowledge holds infinitely often, or request holds finitely often).

Simulation relations

Two states x and y of a Kripke structure are said to be *bisimular* if:

1. x and y agree on the atomic propositions,
2. every successor of x is bisimular to a successor of y and
3. every successor of y is bisimular to a successor of x.

Two states are bisimular if and only if they satisfy the same set of CTL formulas [BCG87]. If $(a_1, a_2, \ldots, a_k)$ are the atomic propositions, then the bisimulation relation can be expressed in the Mu-Calculus as follows:

$$\begin{aligned} \mathit{Bisim} \;=\; \nu Y.\ \lambda x, y.\ & \bigwedge_{1 \le i \le k} (a_i(x) \iff a_i(y)) \\ & \wedge \forall x'.\ (R(x, x') \Rightarrow \exists y'.(R(y, y') \wedge Y(x', y'))) \\ & \wedge \forall y'.\ (R(y, y') \Rightarrow \exists x'.(R(x, x') \wedge Y(x', y')))) \end{aligned}$$

where we have, as usual, identified each atomic proposition with the set of states in which it is true. There is also an asymmetric notion of simulation – we say that a state x simulates a state y if:

1. x and y agree on the atomic propositions,
2. every successor of y is simulated by a successor of x.

If state x simulates state y, then y satisfies every formula satisfied by x in a dialect of CTL called ∀-CTL, which allows only universal path quantifiers [GL91].[16] Testing bisimulation and simulation relations can be used as a form of verification, or it can be used to test abstractions used in compositional model checking techniques [CLM89a, GL91]. The same idea can easily be extended to systems with labeled transitions.

[16]In fact, this is also true for CTL*, an extension of CTL which allows unrestricted linear temporal formulas to be preceded by path quantifiers.

Language containment

The Mu-Calculus can can express the relation of language containment between two deterministic ω-automata. For the sake of simplicity, we consider only deterministic Büchi automata, which are not complete for the class of ω-regular languages, but it is not substantially more difficult to handle more general classes of deterministic automata, such as Street automata.

A finite deterministic Büchi automaton consists of a set of states K, an initial state $p_0 \in K$, an alphabet Σ, a set of transitions $\Delta \subseteq K \times \Sigma \times K$, and an acceptance set $B \subseteq K$. The transition relation is such that, for any state p and symbol σ, there is exactly one q for which $\Delta(p, \sigma, q)$. The automaton accepts an infinite sequence $\sigma \in \Sigma^\omega$ iff the sequence of states p, where $\Delta(p_i, \sigma_i, p_{i+1})$ holds for all i, passes through the acceptance set B infinitely often. The set of sequences accepted by an automaton M is called the language of M and denoted $\mathcal{L}(M)$.

To determine whether the language of a Büchi automaton M is contained in the language of a Büchi automaton M' (with the same alphabet), we define a Kripke structure representing the product of M and M', and write a formula in CTL which is true if and only if every sequence accepted by M is also accepted by M' [CDK90]. This formula can be evaluated using its Mu-Calculus characterization.

The product is defined by its transition relation R, and set of initial states S_0. Let

1. $R = \lambda s, s', r, r'.\ \exists \sigma.\ (\Delta(s, \sigma, r) \wedge \Delta'(s', \sigma, r'))$,
2. $S_0 = \lambda s, s'.\ ((s = p_0) \wedge (s' = p'_0))$,

There is a sequence in the language of M but not in the language of M' if and only if there is an path of the product passing through B infinitely often, but not through B' infinitely often. Thus, $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ iff

$$S_0 \Rightarrow AGA_{\{\lambda s, s'.\ B(s)\}} F \lambda s, s'.\ B'(s')$$

Another possible approach to the language containment problem makes use of the transitive closure of the transition relation. First, we remove from the product structure all transitions that begin or end with a state in B'. That is, let

$$T = \lambda s, s', r, r'[R(s, s', r, r') \wedge \neg B'(s') \wedge \neg B'(r')]$$

The transitive closure of this relation is

$$T^+ = \mu Q[\lambda s, s', r, r'[T(s, s', r, r') \vee \exists u, u'[Q(s, s', u, u') \wedge Q(u, u', r, r')]]]$$

This is the set of all pairs (x, y) of states of the product such that x can reach y without passing through B'. This holds for the pair (x, x) if and only if x is on a cycle not passing through B'. If there is any such x in B, and x is reachable, then there is a path passing through B but not B' infinitely often, hence there is a sequence in $\mathcal{L}(M)$, but not in $\mathcal{L}(M')$. The converse is also true. Hence, $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ if and only if $\neg EF \lambda s, s'.\ (T^+(s, s', s, s') \wedge B(s))$. The EF operator can also be evaluated using the transitive closure, since

$$EFp = \lambda x.\ (p(x) \vee \exists y.\ (R^+(x, y) \wedge p(y)))$$

2.6.2 Symbolic algorithm

By devising a symbolic model checking procedure for the Mu-Calculus, we can quickly establish symbolic algorithms for all of the above properties. If we assume that the domain is B^k, a symbolic model checking algorithm is easily established, by translating formulas into a Boolean Mu-Calculus where the domain is just $B = \{\text{false}, \text{true}\}$. This is done by replacing every individual symbol a by a k-tuple of individual symbols $(a_1, a_2, \ldots a_k)$. Thus, every n-ary term translates to a kn-ary term. In the Boolean Mu-Calculus we can represent terms by Boolean formulas by introducing a new set of dummy individual symbols $d_1, d_2, \ldots$ to represent relational parameters. An n-ary term τ is represented by a formula $\tau[\psi]$ such that

$$(x_1, \ldots, x_n) \in \tau[\phi, \psi]$$
$$\text{iff}$$
$$\tau[\psi](a \leftarrow \phi(a), b \leftarrow \phi(b), \ldots)(d_1 \leftarrow x_1, \ldots, d_n \leftarrow x_n)$$

Given ψ, we can compute the formula representing a term in the Boolean Mu-Calculus by recursion over its structure, as follows:

1. The value of relational variable A is $\psi(A)$.

2. The logical connectives and quantifiers are evaluated by the corresponding QBF operations.

3. The value of an n-ary term $\lambda(v_1, \ldots, v_n).\ \tau$ is

$$\tau[\psi](v_1 \leftarrow d_1, \ldots, v_n \leftarrow d_n)$$

   .

4. The value of the proposition $\tau(v_1, \ldots, v_n)$, where τ is an n-ary term, is $\tau[\psi](d_1 \leftarrow v_1, \ldots, d_n \leftarrow v_n)$.

5. The n-ary relational terms $\mu Y.\ \tau$ and $\nu Y.\ \tau$ are evaluated using the standard fixed point algorithm.

Because the variables are Boolean valued, we can implement all of the above using the operations of QBF, with OBDDs as our representation. The symbolic Mu-Calculus model checking algorithm is shown in pseudo-code form in figure 2.15. Using this algorithm, any quantity that can be characterized in the Mu-Calculus can be computed using the symbolic model checking technique, with the possibility that a combinational explosion can be reduced or avoided. This also allows us to use the expressive powers of the Mu-Calculus in describing and manipulating symbolic algorithms, with the understanding that the translation from Mu-Calculus to a symbolic algorithm is merely mechanical.

2.7 Computing equivalence relations

In this section, we consider the problem of computing a symbolic representation of the equivalence relation between the states of two finite state machines, or between states of the same machine. In the former case, the relation can be used to determine the equivalence of the two machines, while in the latter case, as Lin *et al.* have observed [LTN90], the self equivalence relation can be used in optimizing the logic or register usage of the machine.

```
**function** eval($\tau, \psi$)
    **case**
        $\tau$ a relational variable: **return** $\psi(\tau)$
        $\tau = \neg p$: **return** $\neg$eval($p, \psi$)
        $\tau = p \vee q$: **return** eval($p, \psi$) $\vee$ eval($q, \psi$)
        $\tau = \exists w.\ q$: **return** $\exists w.$ eval($p, \psi$)
        $\tau = \mu Y.\ p$: **return** fixedpoint($Y$,$p$,$\psi(Y \leftarrow$ false))
        $\tau = \nu Y.\ p$: **return** fixedpoint($Y$,$p$,$\psi(Y \leftarrow$ true))
    **end case**
**end function**

**function** fixedpoint($Y$,$p$, $\psi$)
    $Y' =$ eval($p, \psi$)
    **if** $Y' = \psi(Y)$ **then return** $Y'$
    **else return** fixedpoint($Y$,$p$,$\psi(Y \leftarrow Y')$)
**end function**
```

Figure 2.15: Symbolic Mu-Calculus model checking algorithm

2.7.1 State equivalence

We use a standard notion of the equivalence of states of finite Mealy machines. Two states are equivalent if and only if for all input sequences, they yield the same output sequence. The following is an alternate characterization: equivalence is the greatest relation between states such that if x is equivalent to y, then for all inputs, the output in state x is equal to the output in state y, and the successor state of x is equivalent to the successor state of y. Let $\delta(x,z)$ be the function which determines the next state, as a function of current state x and current input z, and let $\gamma(x,z)$, be the function that determines the current output. In the Mu-Calculus, the equivalence relation R_ω is

$$R_\omega = \nu R.\ \lambda x, y.\ \forall z.\ (\gamma(x,z) = \gamma(y,z) \wedge R(\delta(x,z), \delta(y,z))) \tag{2.22}$$

Using the standard fixed point approach, we can evaluate this relation by a sequence of approximations $R_0, R_1, \ldots$, where R_i is the set of state pairs which are equivalent for all input sequences of length i. This sequence is characterized by the recurrence

$$R_1 = \lambda x, y.\ \forall z.\ (\gamma(x,z) = \gamma(y,z)) \tag{2.23}$$

and

$$R_{i+1} = \lambda x, y.\ \forall z.\ (R_i(x,y) \wedge R_i(\delta(x,z), \delta(y,z))) \tag{2.24}$$

This is simply the standard $O(n^2)$ algorithm for computing state equivalence of Mealy machines. The problem of determining whether two Mealy machines are equivalent in their initial states can be approached in two ways – either their equivalence relation can be computed, or the state space of their product can be exhausted by a forward search. The number of iterations required for the former approach can be substantially less, however. In the trivial case of an n-bit counter, the number of iterations in the forward search is is exponential in n, while one step suffices to reach a fixed point in the equivalence calculation, since all states are distinguished by their outputs.

It is immediately seen that the crucial step in calculation 2.24 is the substitution of vector functions $\delta(x,z)$ and $\delta(y,z)$ into R_i. The most obvious way to accomplish this is to use Bryant's *Compose* algorithm. Some other possible methods are introduced in this section. Computing the OBDD representation for a composition of functions is an NP-hard problem (cf., section 2.3.4), thus we expect no good general solutions to the problem. Another tractability issue is whether the approximations to the equivalence relation can be compactly represented using OBDDs. There is no guarantee of this, of course, but there is some reason to believe, *a priori*, that it may often be the case. First of all, for single-machine equivalence, if all distinct states are distinguishable, then the equivalence relation is the identity relation, which can be represented as by a linear size OBDD, provided the component variables of x and y are interleaved in the variable ordering. It also seems plausible that the equivalence relation will often be simply a logical conjunction of independent relations, each corresponding to some modular component of the system. In this case, the OBDD representation will also be compact, provided the variable ordering conforms to the modular structure of the machine. In any case, we will see examples of fairly complex machines whose equivalence relations are expressed compactly in OBDD form.

Algorithm using restrictions

Because of the basic difficulty of computing compositions of OBDDs, it is useful to have some restrictions on the result in order to be able to solve the problem. Fortunately, the decreasing nature of the series of approximations defined in 2.24 provides a constraint on the result of the substitution, since each approximation R_{i+1} is strictly contained in R_i. We can use this fact by rewriting 2.24 as

$$R_{i+1} = \lambda x, y.\ \forall z.\ (R_i(x,y) \wedge (R_i(\delta(x,z), \delta(y,z))\ \downarrow\ R_i)) \tag{2.25}$$

where $\downarrow$ represents the *Restrict* operator introduced by Coudert, Madre and Berthet [CBM89]. This operation produces a function which agrees with $R_i(\delta(x,z), \delta(y,z))$ over the set R_i, attempting to minimize the OBDD size. The restriction can be used to varying advantage, depending on the algorithm used for substitution.

Iterative abstraction algorithm

Another way to provide a restriction on the equivalence relation is first to find the equivalence relation of an abstracted machine. We choose the abstraction in such a way that the equivalence relation of the abstract machine is strictly weaker than the equivalence relation of the original machine. Thus, we can compute the equivalence relation of the abstract machine first, and use it as a restriction in computing the equivalence relation of the original machine. In particular, we can abstract the machine by choosing a subset V of the state variables, and at each approximation, quantifying out the remaining variables existentially. That is, let V^c be the complement of V, and let

$$R_1^V = \lambda x, y.\ \exists V^c.\ \forall z.\ (\gamma(x,z) = \gamma(y,z)) \tag{2.26}$$

and

$$R_{i+1}^V = \lambda x, y.\ \exists V^c.\ \forall z.\ (R_i^V(x,y) \wedge (R_i^V(\delta(x,z), \delta(y,z)) \downarrow R_i^V)) \tag{2.27}$$

It is trivial to see that each approximation in the series R^V is strictly weaker than the corresponding approximation in R. It follows that R_ω^V, the greatest fixed point, is weaker than R_ω. Therefore, we can restrict the entire calculation of R_ω to only those state pairs satisfying R_ω^V. In addition, we can use a series of subsets $V_1 \subset V_2 \subset \cdots \subset V_k$ where V_k is the set of all state variables, restricting the first approximation in each series R^{V_i} to the equivalence relation for the previous subset. Thus, we let

$$R_1^{V_j} = R_\omega^{V_{j-1}} \wedge \lambda x, y.\ \exists V_j^c.\ \forall z.\ (\gamma(x,z) = \gamma(y,z)) \tag{2.28}$$

and

$$R_{i+1}^{V_j} = \lambda x, y.\ \exists V_j^c.\ \forall z.\ (R_i^{V_j}(x,y) \wedge (R_i^{V_j}(\delta(x,z), \delta(y,z)) \downarrow R_i^{V_j})) \tag{2.29}$$

We will refer to this as the iterative abstraction algorithm for computing the equivalence relation. By adding only a few variables to each successive subset, we can in some cases obtain fairly strong restriction, which allows the substitution to be computed more efficiently. In other cases the equivalence relation obtained for the abstracted machine may be trivial, since abstracting out the variables in V^c may result in all machine states appearing equivalent at the outputs. This is especially likely if the abstracted variables hold important control information that enables machine registers to be observed at the outputs. Nonetheless, we can show cases where this incremental approach is greatly more efficient than the basic algorithm.

2.7.2 Methods for functional composition

This section considers methods for substituting functions for variables in OBDDs. This operation is referred to by Bryant as *Compose*. It is the syntactic mechanism corresponding to functional composition. As such, it has a number of applications apart from finding the equivalence relation of finite state machines, including the evaluation of CTL formulas [BF89b]. Most of the algorithms presented here for this purpose have been modified to take as an extra argument a restriction on the result, in the hope that efficiency can be obtained by combining these two operations. We consider the problem of calculating $f(g_1, \ldots, g_n) \downarrow R$, where f, $g_1, \ldots, g_n$ and R are all Boolean functions.

"bottom-up" substitution

This is the method originally proposed by Bryant for his *Compose* algorithm, but with a restriction on the result. In this method, we view each OBDD node in f as a gate, which computes the function "if v_i then h else l" or equivalently, $(\neg v_i \wedge l) \vee (v_i \wedge h)$. Having substituted the functions $g_1, \ldots, g_n$ for the variables in l and h, we can then compute the result for f using the standard $\vee$ and $\wedge$ operators. The basic bottom-up algorithm is

```
**function** bottom-up($f$,$R$)
  **if** $f$ is a leaf **then** return $f$
  **if** bottom-up($f$,$R$) has already been solved **then** return old solution
  **else** [$f$ is a triple $(v_i, f_l, f_h)$]
    $l =$ bottom-up($f_l, R$)
    $h =$ bottom-up($f_h, R$)
    return $((\neg g_i \wedge l) \vee (g_i \wedge h)) \downarrow R$
**end**
```

Note that the restriction operator is used at each step to simplify the result. Since each subproblem is solved only once, the number of recursive calls to bottom-up is $|f|$.

Domain partitioning

The domain partitioning strategy is so named because it divides the problem into two subproblems by partitioning the domain of the functions $g_1, \ldots, g_n$ according to the value of one of the variables. The operation proceeds in several steps.

First, we observe that if any of $g_1, \ldots, g_n$ are constants, we can immediately substitute these values into f, since substitution by a constant is a linear time operation which can only reduce the size of the OBDD. We use the fact that if $g_i = c$, where c is 0 or 1, then

$$f(g_1, \ldots, g_n) = f(v_i \leftarrow c)(g_1, \ldots, g_n) \tag{2.30}$$

Next, we observe that we can eliminate any argument position on which f does not depend, thus obtaining a smaller problem with the same result. We can determine the set of variables on which f depends in linear time, since f depends on v_i if and only if v_i appears in some node in f.

If at this point the function f has been reduced to a constant, we are done. Otherwise, we split the problem into two cases and recurse. We choose the first variable v_i occurring in $g_1, \ldots, g_n$, and apply Shannon's expansion, obtaining two subproblems

$$\begin{aligned} l &= f(g_1(v_i \leftarrow 0), \ldots, g_n(v_i \leftarrow 0)) \\ h &= f(g_1(v_i \leftarrow 1), \ldots, g_n(v_i \leftarrow 1)) \end{aligned}$$

As in other OBDD algorithms, the result is an OBDD $r = (v_i, l, h)$, provided $l \neq h$, otherwise $r = l = h$. Needless to say, we use a hash table, caching the results of subproblems so that the same subproblem is not solved twice. With caching, the complexity of the algorithm is $O(|f| \times \prod |g_i|)$.

Making use of the restriction R in this algorithm is straightforward. If $R = 0$, the result can be any function at all, so we simply return 0. Each time we partition the problem into subproblems, we also split R into two cases, $R(v_i \leftarrow 0)$ and $R(v_i \leftarrow 1)$. The restriction has the effect of cutting off the recursion each time a 0 leaf is reached in R.

Sequential substitution

This is perhaps the simplest approach to substitution; it transforms a simultaneous substitution problem into a sequence of substitutions. This is done by replacing each variable v_i in the OBDD for f by a new variable v'_i. Having done this, it is safe to perform the substitutions of each function g_i for v'_i in any order, since none of the functions $g_1, \ldots, g_n$ depends on any variable being substituted. Substitution of a function for a single variable can be accomplished as follows:

$$f(v'_i \leftarrow g_i) = \exists v'_i.\ [(v'_i \iff g_i) \land f] \tag{2.31}$$

This approach can also make effective use of a restriction. The restriction operator operator may in fact be applied after each substitution step if desired, potentially reducing the size of the intermediate results. In the case of the iterative abstraction algorithm, the fact that some of the variables in the result will later be quantified out existentially can also be put to use. We can move the existential quantifiers for these variables inside the conjunction, thus quantifying the abstracted variables out of the term $(v'_i \iff g_i)$ before applying the conjunction. This may weaken our result somewhat, since $[\exists x.\ a] \land [\exists x.\ b]$ is weaker than $\exists x.[a \land b]$, but it can produce significant reductions in the size of the intermediate results. The final result of the equivalence algorithm is unchanged, since it is computed with no variables abstracted.

2.7.3 Experimental results

This section presents the results of applying the various equivalence relation algorithms to several example state machines, with a range of complexity. The results are compared to published results for the same circuits by Touati *et al.* (computing only the reachable states) and Lin *et al.* (computing the equivalence relation). In all cases, it is self-equivalence that is calculated. It would be interesting to have some results in this section on calculating the state equivalence relation between two different implementations of a given machine, but unfortunately, such examples were lacking. The three different approaches to OBDD substitution are compared, for each example. Where possible, the direct algorithm is used, otherwise, the iterative abstraction

machine	mtd	result (nodes)	b-u (secs)	d-p (secs)	seq (secs)	Touati (secs)	Lin (secs)
sbc	iter	361	2054	$> 10K$	2415	2903	
cpb32	iter	95	45.5	14.4	54.4	14.1	12.10
key	iter	167	342	1738	884	5706	175.20
minmax10	dir	89	197	255	204		
minmax20	dir	59	3.0	4.5	6.7		
minmax30	dir	89	6.2	9.0	15.7		

Table 2.1: Equivalence calculation times

algorithm is used. For example, the state equivalence relation for the machine sbc was calculated using iteration, but could not be calculated directly. Table 2.1 gives the total execution time in seconds, while table 2.2 gives the total number of OBDD nodes used.[17] The latter numbers are not very reliable, since they depend to some extent on arbitrary choices about when to scavenge unused cells and cache entries. However, if the available memory limit of 190,000 nodes is exceeded, it is certain that all of the nodes in use were necessary for the computation, since all available nodes were scavenged when the memory limit was reached. The columns give the following information: the name of the circuit, the method used (direct or iterative), the size of the equivalence relation, and the time or space needed for each of the three substitution methods (bottom-up, domain partitioning, and sequential). The times are for a LISP implementation running on a 1 MIP minicomputer. The final two columns give the results obtained by Touati *et al.* and Lin *et al.* for the same circuit. These times are for C language implementations running on a DEC 5400 and IBM R6000 respectively.

It would seem that for the circuits cpb, key and minmax, which have regular structures with no control registers, there is no clear choice as to

[17]Actually, function graphs with negated arcs were used for this calculation [Bil87], hence the number of nodes may be slightly smaller than what would be obtained using OBDDs.

machine	mtd	result (nodes)	b-u (nodes)	d-p (nodes)	seq (nodes)
sbc	iter	361	22184		34609
cpb32	iter	95	8202	4225	11295
key	iter	167	13328	24868	11563
minmax10	dir	89	16589	17815	17566
minmax20	dir	59	8538	8492	9190
minmax30	dir	89	11952	11886	10001

Table 2.2: Equivalence calculation space

which substitution algorithm is best. The bottom-up algorithm tends to provide the best performance with the least memory usage, but there are a number of exceptions. The machine sbc, which is somewhat more complex, is a more interesting case. Here bottom-up and sequential both provide fairly efficient solutions, although the iterative method was required in both cases to solve the problem. The domain partitioning approach fails to terminate after 10,000 seconds. In the first stage of the iterative algorithm, domain partitioning produced over 100,000 subproblems for a final result of approximately 100 nodes. Obviously, many different subproblems with identical results are being solved. The difficulty is that there is no easy way to identify equivalent problems. It is worth mentioning the the limit on the size of the cache for this method was 5000 entries. With an unbounded cache, the performance of the algorithm may be much better (a matter of theoretical interest at best, since an unbounded cache cannot be provided). It should also be noted that the results for minmax are somewhat anomalous, since the 10-bit version seems to be substantially more complex than the 20- and 30-bit cases. This is explained by the fact that the output functions of these different versions were not the same. In the 20- and 30-bit versions, the outputs appear to depend only on the "last" register, and not the "min" and "max" registers. It is also interesting to observe that for minmax10, not all of the states are distinguishable, that is, the equivalence relation is not the identity.

Comparing these results to those of Touati *et al.*, it is interesting to note that the self-equivalence relation can be computed in less time than the reachable states for sbc and key (taking into account the difference in machines speeds of roughly a factor 10, the equivalence method seems to be about one order of magnitude faster for sbc, and two orders of magnitude faster for key). Of course, the information obtained by the two methods is not the same. It seems, however, that in some cases where the set of reachable states is not obtainable, the equivalence computation may still provide useful information for logic optimization. The results of Lin *et al.* seem to be roughly comparable for the machines key and cpb32 (again, taking into account the difference in machine speeds). It is not clear from the Lin *et al.* article which substitution method was used, since two were mentioned. The one benchmark for which the iterative method was required to produce a result was sbc, but unfortunately Lin *et al.* do not report a figure for this machine. Also, because of the fact the the 20- and 30-bit versions of minmax had modified output functions, it is not possible to compare figures for this benchmark. As a result of these ambiguities, it difficult to draw conclusions about the effectiveness of the iterative abstraction method, except to say that in one case (sbc) it was the only method that successfully computed the equivalence relation.

2.8 Related research

The author first experimented with the use of OBDDs to represent sets of states and transition relations in 1987, building the first symbolic model checker for CTL. Various heuristic improvements to the basic technique were developed, including the OBDD algorithm combining existential quantification and conjunction (cf. section 2.3.4), and the technique of early quantification for disjunctive transition relations (cf. section 2.4.2). Extending this work, Burch, Clarke, Long, McMillan, Dill and Hwang described a symbolic model checking procedure for the propositional Mu-Calculus, which could be used for a variety of purposes, including CTL model checking, testing various process equivalences, testing language containment of ω-automata, and checking satisfiability of LTL formulas [BCM+90]. In 1989, the author used the model checking technique to verify the cache consistency protocol of the Encore Gigamax multiprocessor (see chapter 4). In the process, a model checking system called SMV was developed, along with an associated description language (see chapter 3).

In 1989, the idea of using the OBDD representation for verification of finite state machines appears to have been independently developed by Coudert, Madre and Berthet [CBM89], who used it in their PRIAM system for testing equivalence of finite state machines. They represent a finite state machine by a pair of vector Boolean functions. The function $\delta(v, w)$ yields the next state vector as a function of the current state vector v and the input vector w. The function $\lambda(v, w)$ yields the output vector as a function of v and w. The equivalence of two state machines is tested by creating a combined machine in which both machines receive the same input vector, and the output is a single bit which is true if and only if the output vectors of the two machines are equal. The reachable states of this combined machine are computed. If in all reachable states the output is true, the two machines are equivalent, since no input sequence can produce differing output sequences from the two machines.

The set of reached states is computed as the limit of an increasing series of approximations, starting with the initial state. The set of states reachable in one step from a set S is computed by a function called *Imag*, where $\mathit{Imag}(\delta, S) = \{s \mid \exists v, w : \ v \in S, \ \delta(v, w) = s\}$. Most of Coudert, Madre and Berthet's efforts are applied to computing the *Imag* function without resort to representing the transition relation as an OBDD, which they claim is generally intractable. Their approach begins by reducing the problem of computing the image of a set via a function, to computing the range of a function. This is done using an OBDD operation called *Constrain*. The *Constrain* operator takes two Boolean functions f and g, and returns a function $f' = \mathit{Constrain}(f, g)$ with the following property: for all x', $f'(x') = f(x)$, where x is the nearest Boolean vector to x' (according to a suitable distance metric) such that $g(x) = 1$. If we let $\delta' = \mathit{Constrain}(\delta, S)$, then the image of S via δ is just the range of δ'.

Coudert and Madre suggest two methods for computing the range of δ'. The first is called range partitioning. In this approach, we pick the lowest remaining variable in the ordering (call it v_i), and, and divide the problem into two subproblems, depending on the output of function

δ'_i. Thus,

$$\begin{aligned}(\mathit{Range}(\delta'))(v_i \leftarrow 0) &= \mathit{Range}(\mathit{Constrain}(\delta', \neg\delta'_i))\\(\mathit{Range}(\delta'))(v_i \leftarrow 1) &= \mathit{Range}(\mathit{Constrain}(\delta', \delta'_i))\end{aligned}$$

Note that for any function f,

$$\begin{aligned}\mathit{Constrain}(f, f) &= 1 \text{ and}\\\mathit{Constrain}(f, \neg f) &= 0\end{aligned}$$

so each recursion effectively eliminates one component function of δ'. The recursion terminates when all of the components of δ' are constants.

The other approach, called domain partitioning, is to divide into subproblems based on the value of one of the inputs to δ'. Thus,

$$\mathit{Range}(\delta') = \mathit{Range}(\delta'(v_i \leftarrow 0)) \vee \mathit{Range}(\delta'(v_i \leftarrow 1))$$

Again, the recursion terminates when all of the components of δ' are constants.

Both of these strategies are special cases of a general strategy where one chooses a *cover*, which is a pair of functions h_1 and h_2 such that $h_1 \vee h_2 = 1$, and then computes the recursion

$$\mathit{Range}(\delta') = \mathit{Range}(\mathit{Constrain}(\delta', h_1)) \vee \mathit{Range}(\mathit{Constrain}(\delta', h_2))$$

In the case of range partitioning, $h_1 = \delta'_i$ and $h_2 = \neg\delta'_i$. In the case of domain partitioning, $h_1 = v_i$ and $h_2 = \neg v_i$. It is suggested that other covers may be useful as well. As with other OBDD techniques, a table of pairs $(\delta', \mathit{Range}(\delta'))$ is kept to avoid solving the same subproblem twice. This table is not as effective as the in the case of the standard OBDD operations, however, since the number of possible subproblems is exponential in the number of state variables. Coudert and Madre suggest several optimizations for increasing the hit rate in this table.

A further optimization introduced by Coudert and Madre is to use an OBDD function called *Restrict* to reduce the size of the reached state set before applying the *Imag* operator. The *Restrict* operator takes two functions f and g, and produces a function $f' = \mathit{Restrict}(f, g)$ such that for all x, if $g(x) = 1$,then $f'(x) = f(x)$, otherwise $f'(x)$ is arbitrary.

Usually (but not always), the size of f' is less than the size of f. We note that if R_i is the set of states reachable after i steps of the machine, then

$$\begin{aligned} R_{i+1} &= R_i \vee \mathit{Imag}(\delta, R_i) \\ &= R_i \vee \mathit{Imag}(\delta, \mathit{Restrict}(R_i, \neg R_{i-1})) \end{aligned}$$

As a result, the size of the arguments of *Imag* can sometimes be reduced using *Restrict*.

Coudert and Madre report experimental results for computation of the set of reachable states for a variety of small sequential circuits (mostly ISCAS[18] sequential benchmark circuits). Computing the set of reached states can be useful for generating test patterns or "don't care" conditions for logic optimization [TSL$^+$90]. Unfortunately, Coudert and Madre do not use their techniques to actually test the equivalence of two state machines, so it is unknown whether the technique is useful for this purpose. They have not studied the asymptotic performance of their techniques for classes of circuits, so it is not possible to determine whether their optimizations yield asymptotic improvements.

A variant on the symbolic model checking technique for CTL was proposed by Bose and Fisher [BF89b]. Their technique, which is limited to deterministic finite state machines, also represents the transition relation of the machine by a vector of Boolean functions δ, and uses Bryant's *Compose* operation to compute $EXp = p(v_i \leftarrow \delta_i)$. They do not report experimental results using this technique for practical circuits. A similar technique was proposed by Coudert and Madre [CMB91].

Other researchers have proposed techniques to avoid constructing the transition relation. For example, Burch, Clarke and Long use early quantification (cf. section 2.4.2) for both disjunctive and conjunctive transition relations [BCL91b, BCL91a]. They use the term "partitioned transition relations" for this. The technique is somewhat limited in the case of conjunctive transition relations, because existential quantification only distributes over conjunction in the special case when one of the conjuncts does not depend on the variable being quantified. Nev-

[18]International Symposium on Circuits and Systems

ertheless, there are cases where the support of the component relations is sufficiently disjoint to make this technique effective.

The basic technique is the following: assume we wish to compute $\exists v.\ \bigwedge_i f_i$, where $v = (v_1, \ldots, v_k)$ is a vector of variables and $f = (f_1, \ldots, f_m)$ is a vector of Boolean functions. Since conjunction is associative and commutative, we can combine these functions in any order we choose. In addition, if at any time there is a variable occurring in only one function, we can quantify that variable out, since $\exists w.\ (p \wedge q)$ is equivalent to $(\exists w.p) \wedge q$ when q does not depend on w. Since quantification tends to reduce OBDD size by reducing the number of variables, the strategy is to combine the functions in such an order that variables can be quantified out as soon as possible.

Burch Clarke and Long use a fixed order determined by the user for combining the functions. They show that this is quite effective for pipelined data path circuits, and an asynchronous stack circuit, improving the asymptotic performance as the circuit size increases. For the DME circuit, the asymptotic performance of this method was not as good as a method using a disjunctive transition relation, but it can be more efficient for small rings.[19] It was found most efficient to group the components of the transition relation and combine each group in advance, thus avoiding some computation at each step.

For disjunctive transition relations (interleaving models), Burch, Clarke and Long introduce a modified search order that tends to reduce the representation of the reached state set. In a breadth first search, the representation of this set is complicated by the fact that the after n steps, the number of steps taken by each process is constrained to sum to n. This produces an artificial correlation between the states of otherwise independent processes (cf. section 2.4.2). To counter this, one can modify the search order, searching first all of the states reachable by transitions of one subset of the system processes, then the next, and repeating this process until a fixed point is reached. This technique, called “modified breadth first search”, was effective in reducing the OBDDs representing the reached state sets for an asynchronous stack circuit, but was found not to be as effective as the “conjunctive partitioning” method. For the DME circuit, the modified breadth first

[19]Personal communication.

search method was faster up to about 16 cells, but had slower asymptotic performance. The grouping of processes into subsets was manual.

Another OBDD based technique for computing the reachable states of a machine was introduced by Touati *et al.* [TSL$^+$90]. They also use a conjunction of component relations to represent the transition relation, along with early quantification. However, they combine this technique with the *Constrain* operation of Coudert *et al.*. This reduces the problem of computing the image of a set via a relation to that of computing the codomain of a relation. A series of approximations A_i to the reachable states is computed, such that

$$A_{i+1} = A_i \vee \lambda y.\ \exists x.\ (\bigwedge_j \mathit{Constrain}(R_j, A_i))(x, y)$$

where R is a vector of component relations, each relation determining the new state of one state variable. Touati *et al.* find this technique to be superior to using the transition relation directly and to using the *Imag* operation of Coudert *et al.* for computing the reachable states of the benchmark circuits minmax and sbc, somewhat slower for key, and roughly the same for cpb.32.4. It would be interesting to know for the cases where an improvement was obtained, how much was due to the use of *Constrain* and how much to the use of early quantification. Touati *et al.* have also suggested partitioning complex next-state functions into the composition of a sequence of smaller functions. This could be useful for circuits containing multipliers, or other functions which have no compact OBDD representation.

Touati, Brayton and Kurshan report a technique for testing language containment of ω-automata using OBDDs [TBK91]. They use the L-automaton model of Kurshan [Kur86], and an algorithm similar to the one described in section 2.6.1 using the transitive closure of the transition relation. No experimental results using this technique are available.

Another way that one can test equivalence of two finite state machines is by computing the equivalence relation on states, as described in section 2.7. Lin *et al.* also describe OBDD based algorithms for computing this relation [LTN90]. A comparison of the methods can be found in section 2.7. Lin *et al.* describe how the equivalence relation can be used for computing "don't care" conditions for logic optimization. In a later paper [LN91], Lin shows how this relation (represented as an OBDD) can be used for state minimization, using an operator which takes an equivalence relation and returns a relation which maps every state onto the least element of its equivalence class.

Bryant and Seger have taken an an approach to formal verification using OBDDs based on symbolic simulation [Bry88, BBS90, BS90]. The symbolic simulator is similar to an ordinary logic simulator, except that the inputs are symbolic values (variables) rather than numeric values, and the outputs are given as symbolic functions in terms of these variables. These functions are represented by OBDDs. The simulation method gains a great deal in efficiency by using an abstract interpretation of the circuit model. This abstraction uses a lattice consisting of the three values 0, 1 and X, where X is the least upper bound of 0 and 1. The circuit operations such as AND and OR are abstracted in such a way as to be monotonic with respect to this lattice. Therefore, the output of the abstract simulation is always an upper bound on the output of the concrete simulation. In many cases, a large number of the inputs and initial values of state variables can be replaced by X without sacrificing the particular circuit property being proved. The art in this technique is to decompose the specification in such a way that each part can be verified using only a small number of symbolic values and X everywhere else. The simulation technique is limited to a logic with only next-time operators. These formulas can be verified using symbolic simulations of finite execution sequences. This rules out proving properties such as liveness, fairness or deadlock freedom, but allows safety properties to be proved using invariants.

Bose and Fisher have demonstrated a technique for using representation functions to verify sequential circuits using OBDDs [BF89a]. A representation function maps each state of the implementation to a state of the specification (which is also a circuit). Symbolic simulation techniques can be used to show a kind of single step equivalence between the implementation and specification *vis à vis* this relation. As in the method of Bryant and Seger, this proof can be decomposed into parts in such a way that each part requires only a small number of symbolic variables, with the remaining circuit nodes initialized to X. Typically, an invariant is also required, since the single step equivalence only holds over the reachable state space of the implementation. This technique is also limited in that it cannot prove liveness or deadlock properties.

Long and Grumberg have introduced an abstraction technique using OBDDs which is more general than simply introducing X values [CGL92]. Their technique uses an OBDD to express the relation between the abstract and concrete domains. The abstract transition relation is automatically derived using OBDD techniques from the concrete transition relation. This can be done in a compositional way to reduce the number of symbolic variables that are required. A variety of abstractions have been put to use in this way. For example, a binary number can be represented by its remainders modulo a set of relatively prime numbers. This has allowed the use of the Chinese remainder theorem to prove the correctness of a multiplier circuit. In another case, a single bit was used to represent whether a given binary number in a circuit is equal to a given symbolic binary value. In this way the entire function of the arithmetic unit was abstracted away, allowing a data pipeline circuit with 64 64-bit registers to be verified. This abstraction technique is quite general, and is closely related to more classical abstraction techniques [Kur87]. The difference is that function graph methods are used to actually compute the abstract transition relation, rather than giving this relation *a priori*.

Chapter 3

The SMV system

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can readily specify a system as a synchronous Mealy machine, or as an asynchronous network of abstract, nondeterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only basic data types in the language are finite scalar types. Static, structured data types can also be constructed. The logic CTL allows a rich class of temporal properties, including safety, liveness, fairness and deadlock freedom, to be specified in a concise syntax. SMV uses the OBDD-based symbolic model checking algorithm to efficiently determine whether specifications expressed in CTL are satisfied.

The primary purpose of the SMV input language is to provide a symbolic description of the transition relation of a finite Kripke structure. Any propositional formula can be used to describe this relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock – a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The SMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in SMV is similar to that of single assignment data flow languages. A program can be viewed as a system of simultaneous equations, whose solutions determine the next state. By checking programs for multiple assignments to the same variable, circular dependencies, and type errors, the compiler insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a hardware description language, or a programming language. The SMV system is by no means the last word on symbolic model checking techniques, nor is it intended to be a complete hardware description language. It is simply an experimental tool for exploring the possible applications of symbolic model checking to hardware verification.

3.1 An informal introduction

Before delving into the syntax and semantics of the language, let us first consider a few simple examples that illustrate the basic concepts. Consider the following short program in the language.

```
MODULE main
VAR
  request : boolean;
  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request : busy;
                   1 : {ready,busy};
                 esac;
SPEC
  AG(request -> AF state = busy)
```

The input file describes both the model and the specification. The model is a Kripke structure, whose state is defined by a collection of state variables, which may be of Boolean or scalar type. The variable `request` is declared to be a Boolean in the above program, while the variable `state` is a scalar, which can take on the symbolic values `ready` or `busy`. The value of a scalar variable is encoded by the compiler using a collection of Boolean variables, so that the transition relation may be represented by an OBDD. This encoding is invisible to the user, however.

The transition relation of the Kripke structure, and its initial state (or states), are determined by a collection of parallel assignments (a system of simultaneous equations), which are introduced by the keyword ASSIGN. In the above program, the initial value of the variable `state` is set to `ready`. The next value of `state` is determined by the current state of the system by assigning it the value of the expression

```
case
  state = ready & request : busy;
  1 : {ready,busy};
esac;
```

The value of a case expression is determined by the first expression on the right hand side of a (`:`) such that the condition on the left hand side is true. Thus, if `state = ready & request` is true, then the result of the expression is `busy`, otherwise, it is the set {`ready,busy`}. When a set is assigned to a variable, the result is a non-deterministic choice among the values in the set. Thus, if the value of `status` is not `ready`, or `request` is false (in the current state), the value of `state` in the next state can be either `ready` or `busy`. Non-deterministic choices are useful for describing systems which are not yet fully implemented (*ie.*, where some design choices are left to the implementor), or abstract models of complex protocols, where the value of some state variables cannot be completely determined.

Notice that the variable `request` is not assigned in this program. This leaves the SMV system free to choose any value for this variable, giving it the characteristics of an unconstrained input to the system.

The specification of the system appears as a formula in CTL under the keyword `SPEC`. The SMV model checker verifies that all possible initial states satisfy the specification. In this case, the specification is that invariantly if `request` is true, then inevitably the value of `state` is `busy`.

The following program illustrates the definition of reusable modules and expressions. It is a model of a 3 bit binary counter circuit. Notice that the module name "`main`" has special meaning in SMV, in the same way that it does in the C programming language. The order of module definitions in the input file is inconsequential.

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
SPEC
  AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := value + carry_in mod 2;
DEFINE
  carry_out := value & carry_in;
```

In this example, we see that a variable can also be an instance of a user defined module. The module in this case is `counter_cell`, which is instantiated three times, with the names `bit0`, `bit1` and `bit2`. The counter cell module has one formal parameter `carry_in`. In the instance `bit0`, this formal parameter is given the actual value 1. In the instance `bit1`, `carryin` is given the value of the expression `bit0.carry_out`. This expression is evaluated in the context of the main module. However, an expression of the form $a.b$ denotes component b of module a, just as if the module a were a data structure in a standard programming language. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`. The keyword `DEFINE` is used to assign the expression `value & carry_in` to the symbol `carry_out`. Definitions of this type are useful for describing Mealy machines. They are analogous to macro definitions, but notice that a symbol can be referenced before it is defined.

The effect of the DEFINE statement could have been obtained by declaring a variable and assigning its value, as follows:

```
VAR
  carry_out : boolean;
ASSIGN
  carry_out := value & carry_in;
```

Notice that in this case, the *current* value of the variable is assigned, rather than the next value. Defined symbols are sometimes preferable to variables, however, since they don't require introducing a new variable into the OBDD representation of the system. The weakness of defined symbols is that they cannot be given values non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not. This may be an advantage or a disadvantage, depending on your point of view.

In a parallel-assignment language, the question arises: "What happens if a given variable is assigned twice in parallel?" More seriously: "What happens in the case of an absurdity, like `a := a + 1;` (as opposed to the sensible `next(a) := a + 1;`)?" In the case of SMV, the compiler detects both multiple assignments and circular dependencies, and treats these as semantic errors, even in the case where the corresponding system of equations has a unique solution. Another way of putting this is that there must be a total order in which the assignments can be executed which respects all of the data dependencies. The same logic applies to defined symbols. As a result, all legal SMV programs are realizable.

By default, all of the assignment statements in an SMV program are executed in parallel and simultaneously. It is possible, however, to define a collection of parallel processes, whose actions are interleaved arbitrarily in the execution sequence of the program. This is useful for describing communication protocols, asynchronous circuits, or other systems whose actions are not synchronized (including synchronous circuits with more than one clock). This technique is illustrated by the following program, which represents a ring of three inverting gates.

```
MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);
SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !input;
```

A *process* is an instance of a module which is introduced by the keyword `process`. The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Because the choice of the next process to execute is non-deterministic, this program models the ring of inverters independently of the speed of the gates. The specification of this program states that the output of `gate1` oscillates (*ie.*, that its value is infinitely often zero, and infinitely often 1). In fact, this specification is false, since the system is not forced to execute every process infinitely often, hence the output of a given gate may remain constant, regardless of changes of its input.

In order to force a given process to execute infinitely often, we can use a *fairness constraint*. A fairness constraint restricts the attention of the model checker to those execution paths along which a given CTL formula is true infinitely often. Each process has a special variable called `running` which is true if and only if that process is currently executing. By adding the declaration

```
FAIRNESS
  running
```

to the module `inverter`, we can effectively force every instance of `inverter` to execute infinitely often, thus making the specification true.

One advantage of using interleaving processes to describe a system is that it allows a particularly efficient OBDD representation of the transition relation. We observe that the set of states reachable by one step of the program is the union of the sets of states reachable by each individual process. Hence, rather than constructing the transition relation of the entire system, we can use the transition relations of the individual processes separately and the combine the results (cf. section 2.4.2). This can yield a substantial savings in space in representing the transition relation.

The alternative to using processes to model an asynchronous circuit would be to have all gates execute simultaneously, but allow each gate the non-deterministic choice of evaluating its output, or keeping the same output value. Such a model of the inverter ring would look like the following:

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate2.output);
  gate3 : inverter(gate1.output);
SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)

MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := 0;
  next(output) := !input union output;
```

The union operator allows us to express a nondeterministic choice between two expressions. Thus, the next output of each gate can be either its current output, or the negation of its current input – each gate can choose non-deterministically whether to delay or not. As a result, the number of possible transitions from a given state can be as high as 2^n, where n is the number of gates. This sometimes (but not always) makes it more expensive to represent the transition relation. The relative advantages of interleaving and simultaneous models of asynchronous systems are discussed in section 2.4.2.

As a second example of processes, the following program uses a variable `semaphore` to implement mutual exclusion between two asynchronous processes. Each process has four states: `idle`, `entering`, `critical` and `exiting`. The entering state indicates that the process wants to enter its critical region. If the variable `semaphore` is zero, it goes to the `critical` state, and sets `semaphore` to one. On exiting its critical region, the process sets `semaphore` to zero again.

```
MODULE main
VAR
  semaphore : boolean;
  proc1 : process user;
  proc2 : process user;
ASSIGN
  init(semaphore) := 0;
SPEC
  AG !(proc1.state = critical & proc2.state = critical)

MODULE user
VAR
  state : {idle,entering,critical,exiting};
ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      1 : state;
    esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
    esac;
FAIRNESS
  running
```

If any specification in the program is false, the SMV model checker attempts to produce a counterexample, proving that the specification is false. This is not always possible, since formulas preceded by existential path quantifiers cannot be proved false by a showing a single execution path. Similarly, subformulas preceded by universal path quantifier cannot be proved true by a showing a single execution path. In addition, some formulas require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state.

In the case of the semaphore program, suppose that the specification were changed to

```
  AG (proc1.state = entering -> AF proc1.state = critical)
```

In other words, we specify that if `proc1` wants to enter its critical region, it eventually does. The output of the model checker in this case is shown in figure 3.1. The counterexample shows a path with `proc1` going to the `entering` state, followed by a loop in which `proc2` repeatedly enters its critical region and the returns to its `idle` state, with `proc1` only executing only while `proc2` is in its critical region. This path shows that the specification is false, since `proc1` never enters its critical region. Note that in the printout of an execution sequence, only the values of variables that change are printed, to make it easier to follow the action in systems with a large number of variables.

Although the parallel assignment mechanism should be suitable to most purposes, it is possible in SMV to specify the transition relation directly as a propositional formula in terms of the current and next values of the state variables. Any current/next state pair is in the transition relation if and only if the value of the formula is one. Similarly, it is possible to give the set of initial states as a formula in terms of only the current state variables. These two functions are accomplished by the TRANS and INIT statements respectively. As an example, here is a description of the three inverter ring using only TRANS and INIT:

```
MODULE main
VAR
  gate1 : inverter(gate3.output);
  gate2 : inverter(gate1.output);
  gate3 : inverter(gate2.output);
SPEC
  (AG AF gate1.out) & (AG AF !gate1.out)
```

```
specification is false

AG (proc1.state = entering -> AF proc1.s... is false:

.semaphore = 0
.proc1.state = idle
.proc2.state = idle

next state:
[executing process .proc1]

next state:
.proc1.state = entering

AF proc1.state = critical is false:

[executing process .proc2]

next state:
[executing process .proc2]
.proc2.state = entering

next state:
[executing process .proc1]
.semaphore = 1
.proc2.state = critical

next state:
[executing process .proc2]

next state:
[executing process .proc2]
.proc2.state = exiting

next state:
.semaphore = 0
.proc2.state = idle
```

Figure 3.1: Model checker output for semaphore example

```
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = 0
TRANS
  next(output) = !input | next(output) = output
```

According to the TRANS declaration, for each inverter, the next value of the output is equal either to the negation of the input, or to the current value of the output. Thus, in effect, each gate can choose nondeterministically whether or not to delay. The use of TRANS and INIT is not recommended, since logical absurdities in these declarations can lead to unimplementable descriptions. For example, one could declare the logical constant 0 (false) to represent the transition relation, resulting in a system with no transitions at all. However, the flexibility of these mechanisms may be useful for those writing translators from other languages to SMV.

To summarize, the SMV language is designed to be flexible in terms of the styles of models it can describe. It is possible to fairly concisely describe synchronous or asynchronous systems, to describe detailed deterministic models or abstract nondeterministic models, and to exploit the modular structure of a system to make the description more concise. It is also possible to write logical absurdities if one desires to, and also sometimes if one does not desire to, using the TRANS and INIT declarations. By using only the parallel assignment mechanism, however, this problem can be avoided. The language is designed to exploit the capabilities of the symbolic model checking technique. As a result the available data types are all static and finite. No attempt has been made to support a particular model of communication between concurrent processes (*eg.*, synchronous or asynchronous message passing). In addition, there is no explicit support for some features of communicating process models such as sequential composition. Since the full generality of the symbolic model checking technique is available through the SMV language, it is possible that translators from various languages, process models, and intermediate formats could be created. In particular, existing silicon compilers could be used to translate high level languages with rich feature sets into a low level form (such as a Mealy machine) that could be readily translated into the SMV language.

3.2 The input language

This section describes the various constructs of the SMV input language, and their syntax.

3.2.1 Lexical conventions

An `atom` in the syntax described below may be any sequence of characters in the set {`A-Z,a-z,0-9,_,-`}, beginning with an alphabetic character. All characters in a name are significant, and case is significant. Whitespace characters are space, tab and newline. Any string starting with two dashes ("`--`") and ending with a newline is a comment. A `number` is any sequence of digits. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below.

3.2.2 Expressions

Expressions are constructed from variables, constants, and a collection of operators, including Boolean connectives, integer arithmetic operators, and `case` expressions. The syntax of expressions is as follows.

```
expr ::
        atom                    ;; a symbolic constant
        | number                ;; a numeric constant
        | id                    ;; a variable identifier
        | "!" expr              ;; logical not
        | expr1 "&" expr2       ;; logical and
        | expr1 "|" expr2       ;; logical or
        | expr1 "->" expr2      ;; logical implication
        | expr1 "<->" expr2     ;; logical equivalence
        | expr1 "=" expr2       ;; equality
        | expr1 "<" expr2       ;; less than
        | expr1 ">" expr2       ;; greater than
        | expr1 "<=" expr2      ;; less that or equal
        | expr1 ">=" expr2      ;; greater than or equal
        | expr1 "+" expr2       ;; integer addition
        | expr1 "-" expr2       ;; integer subtraction
        | expr1 "*" expr2       ;; integer multiplication
        | expr1 "/" expr2       ;; integer division
        | expr1 "mod" expr2     ;; integer remainder
        | "next" "(" id ")"     ;; next value
        | set_expr              ;; a set expression
        | case_expr             ;; a case expression
```

An `id`, or identifier, is a symbol or expression which identifies an object, such as a variable or defined symbol. Since an `id` can be an atom, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the compiler as an error. The expression `next(x)` refers to the value of identifier `x` in the next state (see section 3.2.3). The order of parsing precedence from high to low is

```
        *,/
        +,-
        mod
        =,<,>,<=,>=
        !
        &
        |
        ->,<->
```

Operators of equal precedence associate to the left. Parentheses may be used to group expressions.

A `case` expression has the syntax

```
case_expr ::
        "case"
          expr_a1 ":" expr_b1 ";"
          expr_a2 ":" expr_b2 ";"
          ...
        "esac"
```

A case expression returns the value of the first expression on the right hand side, such that the corresponding condition on the left hand side is true. Thus, if `expr_a1` is true, then the result is `expr_b1`. Otherwise, if `expr_a2` is true, then the result is `expr_b2`, *etc.* If none of the expressions on the left hand side is true, the result of the `case` expression is the numeric value 1. It is an error for any expression on the left hand side to return a value other than the truth values 0 or 1.

A set expression has the syntax

```
set_expr ::
        "{" val1 "," val2 "," ... "}"
        | expr1 "in" expr2      ;; set inclusion predicate
        | expr1 "union" expr2   ;; set union
```

A set can be defined by enumerating its elements inside curly braces. The elements of the set can be numbers or symbolic constants. The inclusion operator tests a value for membership in a set. The union operator takes the union of two sets. If either argument is a number or symbolic value instead of a set, it is coerced to a singleton set.

3.2.3 Declarations

The `VAR` declaration

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation

```
decl :: "VAR"
          atom1 ":" type1 ";"
          atom2 ":" type2 ";"
          ...
```

The type associated with a variable declaration can be either Boolean, scalar, or a user defined module. A type specifier has the syntax

```
type :: boolean
        | "{" val1 "," val2 "," ... "}"
        | atom [ "(" expr1 "," expr2 "," ... ")" ]
        | "process" atom [ "(" expr1 "," expr2 "," ... ")" ]

val  :: atom | number
```

A variable of type `boolean` can take on the numerical values 0 and 1 (representing false and true, respectively). In the case of a list of values enclosed in set brackets (where atoms are taken to be symbolic constants), the variable is a scalar which can take any of these values. Finally, an `atom` optionally followed by a list of expressions in parentheses indicates an instance of module `atom` (cf. section 3.2.4). The keyword `process` causes the module to be instantiated as an asynchronous process (cf. section 3.2.6).

The `ASSIGN` declaration

An assignment declaration has the form

```
decl :: "ASSIGN"
          dest1 ":=" expr1 ";"
          dest2 ":=" expr2 ";"
          ...

dest :: atom
        | "init" "(" atom ")"
        | "next" "(" atom ")"
```

On the left hand side of the assignment, `atom` denotes the current value of a variable, `init(atom)` denotes its initial value, and `next(atom)` denotes its value in the next state. If the expression on the right hand side evaluates to an integer or symbolic constant, the assignment simply means that the left hand side is equal to the right hand side. On the other hand, if the expression evaluates to a set, then the assignment means that the left hand side is contained in that set. It is an error if the value of the expression is not contained in the range of the variable on the left hand side.

In order for a program to be implementable, there must be some order in which the assignments can be executed such that no variable is assigned after its value is referenced. This is not the case if there is a circular dependency among the assignments in any given process. Hence, such a condition is an error. In addition, it is an error for a variable to be assigned more than once simultaneously. To be precise, it is an error if:

1. the next or current value of a variable is assigned more than once in a given process, or

2. the initial value of a variable is assigned more than once in the program, or

3. the current value and the initial value of a variable are both assigned in the program, or

4. the current value and the next value of a variable are both assigned in the program, or

5. there is a circular dependency, or

6. the current value of a variable depends on the next value of a variable.

The `TRANS` declaration

The transition relation R of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a Boolean valued expression, introduced by the `TRANS` keyword. The syntax of a `TRANS` declaration is

```
decl :: "TRANS" expr
```

It is an error for the expression to yield any value other than 0 or 1. If there is more than one `TRANS` declaration, the transition relation is the conjunction of all of `TRANS` declarations.

The `INIT` declaration

The set of initial states of the model is determined by a Boolean expression under the `INIT` keyword. The syntax of a `INIT` declaration is

```
decl :: "INIT" expr
```

It is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one `INIT` declaration, the initial set is the conjunction of all of the `INIT` declarations.

The SPEC declaration

The system specification is given as a formula in the temporal logic CTL, introduced by the keyword `SPEC`. The syntax of this declaration is

```
decl :: "SPEC" ctlform
```

A CTL formula has the syntax

```
ctlform ::
    expr                        ;; a Boolean expression
    | "!" ctlform               ;; logical not
    | ctlform1 "&" ctlform2     ;; logical and
    | ctlform1 "|" ctlform2     ;; logical or
    | ctlform1 "->" ctlform2    ;; logical implies
    | ctlform1 "<->" ctlform2   ;; logical equivalence
    | "E" pathform          ;; existential path quantifier
    | "A" pathform          ;; universal path quantifier
```

The syntax of a path formula is

```
pathform ::
        "X" ctlform                     ;; next time
        "F" ctlform                     ;; eventually
        "G" ctlform                     ;; globally
        ctlform1 "U" ctlform2           ;; until
```

The order of precedence of operators is (from high to low)

```
        E,A,X,F,G,U
        !
        &
        |
        ->,<->
```

Operators of equal precedence associate to the left. Parentheses may be used to group expressions. It is an error for an expression in a CTL formula to contain a `next()` operator or to return a value other than 0 or 1. If there is more than one `SPEC` declaration, the specification is the conjunction of all of the `SPEC` declarations.

The `FAIR` declaration

A fairness constraint is a CTL formula which is assumed to be true infinitely often in all fair execution paths. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths. Fairness constraints are declared using the following syntax:

```
decl :: "FAIR" ctlform
```

A path is considered fair if and only if all fairness constraints declared in this manner are true infinitely often.

The `DEFINE` declaration

In order to make descriptions more concise, a symbol can be associated with a commonly used expression. The syntax for this declaration is

```
decl :: "DEFINE"
          atom1 ":=" expr1 ";"
          atom2 ":=" expr2 ";"
          ...
```

When every an identifier referring to the symbol on the left hand side occurs in an expression, it is replaced by the *value* of the expression on the right hand side (not the expression itself). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

3.2.4 Modules

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be parameterized, so that each instance of a module can refer to different data values. A module can contain instances of other modules, allowing a structural hierarchy to be built. The syntax of a module is as follows.

```
module ::
        [ "OPAQUE" ]
        "MODULE" atom [ "(" atom1 "," atom2 "," ... ")" ]
          decl1
          decl2
          ...
```

The optional keyword `OPAQUE` is explained in the section on identifiers. The atom immediately following the keyword `MODULE` is the name associated with the module. Module names are drawn from a separate name space from other names in the program, and hence may clash with names of variables and definitions. The optional list of atoms in parentheses are the formal parameters of the module. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated.

A *instance* of a module is created using the VAR declaration (cf. section 3.2.3). This declaration supplies a name for the instance, and also a list of actual parameters, which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error if the number of actual parameters is different from the number of formal parameters. The semantics of module instantiation is similar to call-by-reference. For example, consider the following program fragment:

```
...
VAR
  a : boolean;
  b : foo(a);
...
MODULE foo(x)
ASSIGN
  x := 1;
```

The variable `a` is assigned the value `1`. Now consider the following program:

```
...
DEFINE
  a := 0;
VAR
  b : bar(a);
...
MODULE bar(x)
DEFINE
  a := 1;
  y := x;
```

In this program, the value assigned to `y` is 0. Using a call-by-name (macro expansion) mechanism, the value of `y` would be 1, since `a` would be substituted as an expression for `x`.

Forward references to module names are allowed, but circular references are not, and result in an error.

3.2.5 Identifiers

An `id`, or identifier, is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows.

```
id ::
        atom
        | id "." atom
```

An *atom* identifies the object of that name as defined in a `VAR` or `DEFINE` declaration. If a identifies an instance of a module, then the expression $a.b$ identifies the component object named b of instance a. This is precisely analogous to accessing a component of a structured data type. Note that an actual parameter of module instance a can identify another module instance b, allowing a to access components of b, as in the following example:

```
...
VAR
  a : foo(b);
  b : bar(a);
...
MODULE foo(x)
DEFINE
  c := x.p | x.q;

MODULE bar(x)
VAR
  p : boolean;
  q : boolean;
```

Here, the value of `c` is the logical *or* of `p` and `q`. If the keyword `OPAQUE` appears before a module definition, then the variables of an instance of that module are not externally accessible. Thus, the following program fragment is not legal:

```
...
VAR
  a : foo();
DEFINE
  b := a.x;
...
OPAQUE MODULE foo()
VAR
  x : boolean;
...
```

3.2.6 Processes

Processes are used to model interleaving concurrency, with shared variables. A *process* is a module which is instantiated using the keyword `process` (cf. section 3.2.3). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel, simultaneously. Each instance of a process has special variable Boolean associated with it called `running`. The value of this variable is `1` if and only if the process instance is currently selected for execution. The rule for determining whether a given variable is allowed to change value when a given process is executing is as follows: if the next value of a given variable is not assigned in the currently executing process, but is assigned in some other process, then the next value is the same as the current value.

3.2.7 Programs

The syntax of an SMV program is

```
program ::
        module1
        module2
        ...
```

There must be one module with the name `main` and no formal parameters. The module `main` is the one instantiated by the compiler.

3.3 Formal semantics

In this section we assign a formal semantics to SMV programs. In essence, a program is viewed as a system of equations whose solutions determine the transition relation and initial states of a Kripke structure. In fact, this semantics assigns meaning to some programs which are not actually accepted by the compiler due to the rules regarding multiple assignments and circular dependencies. Here, we define a semantics for a subset of the language which does not include the `process` keyword. This subset will be called SMV.0. The semantics of SMV.0 is syntax directed – the denotation of a program is a function of the denotations of its syntactic components. It is also *compositional* with regard to bisimulation and simulation, as we will prove in chapter 5. This makes it possible to use compositional proof methods for verifying SMV.0 programs, including induction over the structure of programs. The semantics for SMV.1, which includes the `process` keyword, is given in appendix A.

3.3.1 The model

The set N of *names*, is the set of all character strings made up of the letters, the digits, the underscore and the minus sign characters, beginning with a letter. The *store* $L = L_V \cup L_H$ is made up of two disjoint, countably infinite sets of *locations* L_V and L_H. We will call the former the *visible* locations, and the latter the *hidden* locations. The set of locations L is defined recursively. It is the least set such that

1. if $n \in N$, then $n \in L_V$, and
2. if $l \in L_V$ and $n \in N$, then $l.n \in L_V$, and
3. if $l \in L_V$, then $.l \in L_H$.

The set of values V is the union of the integers in the range $[-2^{31}, 2^{31}-1]$ and N, the set of names. A *state* $x : L \to V$ is a function from locations to values. Let $S = L \to V$ be the set of all possible states.

If p is a declaration, then its denotation $[\![p]\!]$ is a triple (T, I, R). The T component is a partial function from L to the finite subsets of V.

If l is a location, then $T(l)$, when defined, is the *type* of l – the set of values that can be assigned to location l. The component $I \subseteq S$ is the set of initial states. Finally, the component $R \subseteq S \times S$ is the transition relation.

In the following sections, we define the denotations of the various kinds of declarations. We then define a composition operator $\|$ which gives the denotation of a program in terms of its declarations.

3.3.2 Expressions

An expression denotes a function from states to finite subsets of V, according to the following rules:

1. If v is a value, then $[\![v]\!](x) = \{v\}$.

2. If l is a location, then $[\![l]\!](x) = \{x(l)\}$.

3. If e_1, e_2 are expressions, and o is one of

   `+`, `-`, `*`, `/`, `mod`, `>`, `>=`, `<`, `<=`, `=`, `&`, `|`, `->`, `<->`

   then

   $$[\![e_1\ o\ e_2]\!](x) = \{[\![o]\!](v_1, v_2) \mid v_1 \in [\![e_1]\!](x),\ v_2 \in [\![e_2]\!](x)\}$$

4. If e is an expression, then

   $$[\![!e]\!](x) = \{[\![!]\!](v) \mid v \in [\![e]\!](x)\}$$

5. If e_1, e_2 are expressions,

   $$[\![e_1\ \texttt{union}\ e_2]\!](x) = [\![e_1]\!] \cup [\![e_2]\!]$$

6. If e_1, e_2 are expressions,

   $$[\![e_1\ \texttt{in}\ e_2]\!](x) = [\![e_1]\!] \subseteq [\![e_2]\!]$$

The functions denoted by `+`, `-`, `*`, `/` are the usual functions of arithmetic modulo 2^{32}. The function denoted by `mod` is the positive remainder of division mod 2^{32}. The functions denoted by the relational operators `>`, `>=`, `<` and `<=` return 0 when the relation is false and 1 when the relation is true, and are defined for numeric values only. For non-numeric values, they return $\perp$. The equality operator `=` is defined for all values, and returns 0 when they are unequal, and 1 when they are equal. The functions denoted by the Boolean operators are `&` (for and), `|` (for or), `!` (for not), `->` (for implies) and `<->` (for logical equivalence) are defined only for the values 0 and 1, and return $\perp$ otherwise.

3.3.3 Assignments and definitions

There is no semantic difference between assignments and definitions. If l is a location, and e is an expression, then the assignment $l := e$; denotes a triple (T, I, R), where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid l(x) \in [\![e]\!](x)\}$

The assignment `next`(l) $:= e$; denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid l(y) \in [\![e]\!](x)\}$

The assignment `init`(l) $:= e$; denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = \{x \in S \mid l(x) \in [\![e]\!](x)\}$
3. $R = S^2$

3.3.4 Variable declarations

If l is an identifier and $v_1, v_2, \ldots, v_n$ are values, then

$$\texttt{VAR}\ l\ :\ \{v_1, v_2, \ldots, v_n\};$$

denotes a triple (T, I, R) where

1. $T = \{(l, \{v_1, v_2, \ldots, v_n\})\}$
2. $I = \{x \in S \mid x(l) \in \{v_1, v_2, \ldots, v_n\})\}$
3. $R = \{(x, y) \in S \mid x(l), y(l) \in \{v_1, v_2, \ldots, v_n\})\}$

3.3.5 Renaming

Let $\phi : L \to L$ be a function from locations to locations. This in turn induces a map Φ on states, such that for all states x and locations l,

$$\Phi(x)(l) = x(\phi(l)).$$

If $M = (T, I, R)$, then let $\phi(M) = (T', I', R')$ where

1. $T'(\phi(l)) = T(l)$,
2. $I' = \{x \mid \Phi(x) \in I\}$ and
3. $R' = \{(x, y) \mid (\Phi(x), \Phi(y)) \in R\}$.

Note that the definition of T does not make sense if ϕ maps two locations with different types onto the same location. In this case, $\phi(M)$ is a type error. There are two rules regarding the renaming function ϕ which must be respected to allow compositional reasoning about SMV programs. These are:

1. A hidden location cannot be renamed to a visible location, and
2. Two distinct locations cannot be mapped to the same hidden location.

These rules are respected by the SMV.0 semantics. Notice that it is allowable to rename visible locations to hidden locations. In this way, we can accomplish both hiding and renaming with the same operator.

3.3.6 Parallel composition

The parallel composition of two processes $M_1 \parallel M_2$ is formed in two steps. First, a renaming is applied to map the hidden locations of M_1 and M_2 onto disjoint spaces. Then the union of the type functions T and the intersections of the initial sets I and the transition relations R are taken. Clearly, this does not make sense if the T components do not agree on the type of some location, since the union would not be a function. Formally, let $M_1 = (T_1, I_1, R_1)$ and $M_2 = (T_2, I_2, R_2)$. Let n_1 and n_2 be two distinct names. For $i \in 1, 2$, let $\phi_i(l) = .n_i.l$ for all $l \in L_H$ and $\phi_i(l) = l$ otherwise, and let $M'_i = \phi(M_i)$. The parallel composition $M = M_1 \parallel M_2$ is defined as follows:

1. $T = T'_1 \cup T'_2$
2. $I = I'_1 \cap I'_2$
3. $R = R'_1 \cap R'_2$

If $d_1, d_2, \ldots, d_k$ are declarations, then $[\![d_1\ d_2\ \ldots\ d_k]\!]$ is the parallel composition

$$[\![d_1]\!] \parallel [\![d_2]\!] \parallel \cdots \parallel [\![d_k]\!]$$

.

3.3.7 Instantiation

Suppose that module A is defined as follows:

$$\texttt{MODULE}\ A(n_1, n_2, \ldots, n_k)\ D$$

where $n_1, n_2, \ldots, n_k$ are distinct names and D is a sequence of declarations. Let $r, l_1, l_2, \ldots, l_k$ be visible locations. Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \leq i \leq k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. for all $n \in N - \{n_1, n_2, \ldots, n_k\}$, $\phi(n) = r.n$, and $\phi(n.l) = r.n.l$,
3. $\phi(.l) = .l$

Then $[\![\texttt{VAR}\ r\ :\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

On the other hand, suppose that A is defined as follows:

$$\texttt{OPAQUE MODULE}\ A(n_1, n_2, \ldots, n_k)\ D$$

where $n_1, n_2, \ldots, n_k$ are distinct names and D is a sequence of declarations. Let $r, l_1, l_2, \ldots, l_k$ be visible locations. Let m_1 and m_2 be distinct names in N. Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \leq i \leq k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. for all $n \in N - \{n_1, n_2, \ldots, n_k\}$, $\phi(n) = .m_1.n$, and $\phi(n.l) = .m_1.n.l$,
3. $\phi(.l) = .m_2.l$

Then $[\![\texttt{VAR}\ r\ :\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

3.3.8 Specifications

Each program is associated with a Kripke structure which determines the truth value of CTL formulas in the specification. The atomic propositions in this case are all the Boolean valued expressions. The Kripke structure associated with a program whose denotation is the triple (T, I, R) is a Kripke model $K = (S, R, L')$ where

1. S is the set of states defined above,
2. R is the transition relation, and
3. if e is an expression, then

$$L'(e) = \{x \in S \mid [\![e]\!](x) = \{1\}\}$$

The specification is a formula f in CTL with fairness constraints. It is satisfied exactly when $K, s_0 \models f$ for all $s_0 \in I$.

Chapter 4

A Distributed Cache Protocol

In this chapter, we look at an application of the SMV symbolic model checker to a cache consistency protocol developed at Encore Computer Corporation for their Gigamax distributed multiprocessor [MS91]. This protocol is of interest as a test case for automatic verification for two reasons. First, it is not a theoretical exercise, but a real design, which is driven by considerations of performance and economics, as well as the usual constraints of industrial design, such as compatibility with existing hardware and software. Second, this protocol is a good example of a system where random simulation methods are ineffective in finding design errors.

The Gigamax is a distributed, shared memory multiprocessor, in which the processors are grouped into clusters. Each cluster has a local bus, and uses bus snooping [AB86] to maintain cache consistency within the cluster. In addition, each cluster has an interface called a UIC, which links the cluster into a network. The UIC keeps the caches in the cluster consistent with the rest of the network by acting as both a bus snooper and a bus master on behalf of the remote clusters, using a table which keeps track of the remote status of all cache blocks from the local main memory. This allows it to intervene in bus transactions which affect remotely owned blocks, and to send appropriate invalidation or call back requests to the network. The network is organized into a hierarchy, as depicted in figure 4.1. The global bus, at the top of the

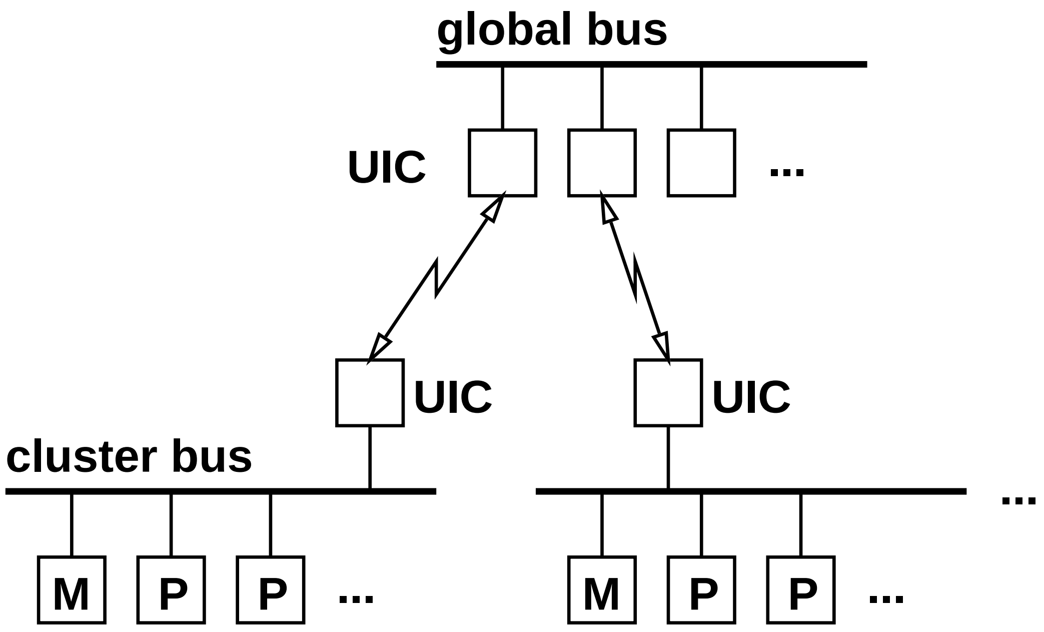


Figure 4.1: Gigamax memory architecture

hierarchy, has one UIC connected to each cluster. These UICs record the status of all cache blocks which are present in the corresponding cluster. This eliminates the need for directory pointers in main memory, at the possible expense of a bottleneck in the global bus.

Protocols such as this are difficult to debug using simulation, in part because the order of events such as cache misses and message arrivals in various parts of the system is unpredictable. Subtle errors sometimes require a long sequence of such events to manifest themselves. Since the number of such sequences is combinatoric, the probability of such a sequence occurring in a random simulation rapidly vanishes as the sequence length increases. Nevertheless, for the design process to stabilize, it is necessary to provide timely information about errors to the design team, since the greater the delay in discovering an error, the greater is the disruption required to fix it. Ideally, a protocol should be error free before a hardware (or software) implementation is considered. Otherwise, the options for fixing the errors will be greatly limited by cost considerations, and the likelihood of the design change introducing other errors will be high.

For this reason, we will consider the verification of the Gigamax protocol at a high level of abstraction, neglecting many admittedly important details of the implementation, such as the widespread use of pipelining, or the link level protocol that communicates messages between clusters. The basic method for building an abstract model of a protocol is to introduce *nondeterminism* in those cases where the level of detail of the model is insufficient to uniquely determine the outcome of an event, or where design decisions have been left open. We will make a note of places in the model where nondeterminism has been used in this way, and in what way the state of an implementation might correspond to the state of the abstract model.

4.1 The Protocol

The purpose of a cache consistency system is to provide the illusion to the programmer of a distributed computer that all processors in the system have access to a shared global store. This illusion must be provided despite the fact that the physical storage is distributed. To reduce the latency of access to the distributed main storage, each processor is provided with a local cache – a semi-associative store, which holds a collection of memory blocks recently used by the processor. The time required to access to this store is less than to access main storage. An access to a memory block stored in the cache is called a *hit*, while an access to a memory block *not* stored in the cache is called a *miss*. A miss requires an access to main storage (which may be remote), to retrieve the required memory block and enter it in the cache. This may result in the *replacement* of another block in the cache, to make room for the block being entered in the cache. If the replaced block has been modified while in the cache, it must be returned to main storage. This is called a *copy back* operation.

The first cache consistency protocols for multiprocessors were called *bus snooping* protocols [AB86]. They required that the processors in the system be connected by a bus, or other broadcast medium. In a bus snooping system, each time a memory access occurs over the bus, all of the caches are checked to determine whether they contain the addressed block. If the block is present in a cache, a change in status may be required. For example, if the block is present and modified, the access must be stalled while the modified data are copied back to main storage. In a more sophisticated protocol, the cache with the modified data may supply the data directly to the requesting cache, without the intermediary of main storage. In case of a memory access caused by an attempt to *modify* the data, all caches in which the block is present must *invalidate*, that is, remove the block from cache storage. This insures that all cached copies of the block remain consistent.

The Gigamax protocol uses bus snooping techniques to maintain consistency of the caches within a single cluster. The main difference between the Gigamax snooping protocol and those described in [AB86] is that the Gigamax uses a *split transaction* bus. This means that a processor accessing memory over the bus first places a request on the bus, and then frees the bus for other transactions while awaiting a response. The bus snooping technique is not practical for large scale multiprocessors, because the broadcast medium quickly becomes saturated. For this reason, the Gigamax uses a message passing protocol to maintain consistency between clusters. The split transaction bus protocol allows traffic to continue on the bus while messages are in transit in the network.

The terminology used in the sequel is changed somewhat from the Encore terminology, and the protocol is somewhat simplified to make the presentation clearer. The basic protocol is preserved, however, including a subtle error which was discovered by the SMV system. The following is a description of the protocol, first in English, then in the SMV input language. In the model, we consider only the status of a single memory block. This is our first use of abstraction, and results in nondeterminism in several places in the model.

4.1.1 Processors

Each memory block stored in each cache has an associated *state*, which can be either *invalid*, *shared*, or *owned*. Alternative names for these states would be *absent*, *present*, and *modified*, respectively. The shared state indicates that there may be other processors which have this block stored in their cache. Therefore, a block in the shared state can be read by the processor, but not written, since writing might result in an inconsistency between two caches. The owned state indicates that no other processors have this block in their cache, and that the data in the cache have been modified. Therefore, a block in the owned state can be both read and written by the processor. The invalid state indicates that the block is not present in the cache. Therefore, the block cannot be read or written by the processor.

```
MODULE cache-device

VAR
  state : {invalid,shared,owned};


DEFINE
  readable := ((state = shared) | (state = owned)) & !waiting;
  writable := (state = owned) & !waiting;
```

The split transaction bus snooping protocol works in the following way. At each bus cycle, the bus arbiter chooses a processor among the requesting processors to be the bus *master*. The remaining processors are referred to as *slaves*. The master issues a *command* on the bus, of which there are three basic types. A *read* command is a request for a given memory block, and is answered by a *response* command. A *write* command stores data in main memory. The write and response commands can be combined into a single command called a *write-response*, which has the simultaneous effect of supplying data to a requester and storing it in main memory. Each command also signals the next state that the bus master will enter. Thus, a *read-owned* command indicates that the bus master intends to modify the data, and a *read-shared* indicates that it does not. A *write-shared* indicates that the bus master is writing data, but maintaining a shared copy, while a *write-invalid* indicates that it is not keeping the block (*eg.*, it is replacing it with another block). The basic commands, and their uses are summarized in table 4.1. We note that no external command is required to go from the shared state to the invalid state. This occurs when a shared block is removed to make room for another block in the cache. Since our model does not contain the states of any other blocks, we allow this replacement to occur nondeterministically, at any time. Thus we model any possible cache replacement policy.

A slave, observing a command on the bus, may decide to modify its state. For example, a slave observing a read-owned command changes its state to invalid, since the bus master, entering the owned state, will assume it has the only cached copy of the block. Correspondingly, a slave in the owned state observing a read-shared command will change to the shared state. A special command called *invalidate* is used to invalidate all caches in the system. A slave observing this command changes to the invalid state.

```
ASSIGN
  init(state) := invalid;
  next(state) :=
    case
      abort : state;
      master :
        case
          CMD = read-shared        : shared;
          CMD = read-owned         : owned;
          CMD = write-invalid      : invalid;
          CMD = write-shared       : shared;
          1 : state;
        esac;
      !master :
        case
          CMD = read-owned        : invalid;
          CMD = invalidate & !waiting : invalid;
          CMD = read-shared & state = owned : shared;
          state = shared & !waiting : {shared,invalid};
          1 : state;
        esac;
    esac;
```

On receiving the command, each slave checks its own cache and indicates the state of the block in its own cache by asserting the signals *reply-owned*, and *reply-waiting* on the bus. These are *wired or* signals, meaning that the signal is observed to be asserted on the bus if one or

from state	command	to state	cause
invalid	read-shared	shared	read miss
invalid or shared	read-owned	owned	write miss
owned	write-invalid	invalid	copy-back
owned	write-resp-invalid	invalid	snoop read-owned
owned	write-shared	shared	write-through
owned	write-resp-shared	invalid	snoop read-shared

Table 4.1: Summary of commands

more caches assert the signal. The reply-owned signal is asserted by a slave when the block is in the owned state in the slave's cache. Reply-waiting is asserted when the slave has previously requested the block, and is waiting for a response. This signal will be discussed in more detail shortly. The process of looking up the slave's state and signaling on the bus is known as bus snooping. On observing a read command, a slave in the owned state sets a flag called *snoop*. This causes the cache to issue a write-response at a later bus cycle, supplying the data to the requester, and simultaneously storing it in main memory. When this happens, the snoop flag is reset.

An additional reply signal called *reply-stall* may be asserted by any slave, including main storage, if the slave if not ready to respond to the command because some resource is busy. If reply-stall is asserted, the command is nullified.

```
DEFINE
  reply-owned := state = owned;

VAR
  snoop : boolean;

ASSIGN
  init(snoop) := 0;
  next(snoop) :=
    case
      abort : snoop;
      state = owned & CMD = read-shared : 1;
      state = owned & CMD = read-owned  : 1;
      CMD = response           : 0;
      CMD = write-resp-invalid : 0;
      CMD = write-resp-shared  : 0;
      1 : snoop;
    esac;
```

After issuing a read command, the master releases the bus and waits for a *response*. During this time, a flag called *waiting* is set. Normally, if no slave asserts reply-owned, the response comes from main memory. If any slave asserts reply-owned, however, main memory is inhibited, allowing the slave to supply the data at a future cycle with a write-response command.

```
MODULE bus-device

VAR
  master  : boolean;
  cmd : {idle,read-shared,read-owned,cty-read,write-invalid,
         write-shared,write-resp-invalid,write-resp-shared,
         invalidate,response};
  waiting : boolean;
  reply-stall : boolean;

ASSIGN
  init(waiting) := 0;
  next(waiting) :=
    case
      abort : waiting;
      master  & CMD = read-shared        : 1;
      master  & CMD = read-owned         : 1;
      CMD = response                     : 0;
      CMD = write-resp-invalid           : 0;
      CMD = write-resp-shared            : 0;
      1 : waiting;
    esac;
```

A slave which is waiting for a given cache block responds to any read command for that block by asserting reply-waiting. This nullifies the read command and forces the master to retry at a later cycle.

```
DEFINE
  reply-waiting := waiting;
  abort := REPLY-STALL
           | ((CMD = read-shared | CMD = read-owned)
              & REPLY-WAITING);
```

The commands which may be issued by a processor when it is bus master are a function of the state. For example, if the snoop flag is set, the processor may issue a write-response on the bus. From the owned state, a processor may issue a write-invalid command in order to replace the cache block with another. A processor in the shared state may issue a read-owned in case of a write miss, and a processor in the invalid state may issue either a read-shared or a read-owned command, in case of a read miss and write miss respectively.

```
MODULE processor(CMD,REPLY-OWNED,REPLY-WAITING,REPLY-STALL,DATA)
ISA bus-device
ISA cache-device

ASSIGN
  cmd :=
    case
      master & snoop & state = invalid : write-resp-invalid;
      master & snoop & state = shared : write-resp-shared;
      master & state = owned & !waiting : write-invalid;
      master & state = shared & !waiting : read-owned;
      master & state = invalid : {read-shared,read-owned};
      1 : idle;
    esac;
```

4.1.2 The local UIC interface

The UIC is the interface from one cluster to another. UICs come in pairs, connected by a communication link. A UIC is said to be *local* for a given memory block if that block is found in main storage on the *same side* of the link as the UIC. It is said to be *remote* if the memory block is found in a main memory on the *other side* of the link. Thus, for any memory block, one of the UICs in the pair is local, and the other remote. The UIC determines whether it is local or remote by address decoding. We consider the local case first. In this discussion, *local* refers to any part of the system on the bus side of the UIC, and *remote* refers to any part of the system on the link side of the UIC.

Viewed from the bus, the UIC behaves like a processor, with the capability to issue and respond to commands. The UIC's cache records the state (but not the data) of all blocks of the local main storage that are present in remote caches. This allows the UIC to snoop the bus on behalf of remote caches. The UIC performs this function in exactly the same manner as the processors. The state of a block in the UIC changes with commands issued in the same manner as the state of cache blocks in processor caches.

The UIC receives command messages from from the link, and stores them in one of two queues. The low priority queue is for read commands, and the high priority queue is for all other commands. The depth of the queues is arbitrary, but for now, we consider queues of only one entry. A command in one of the queues is issued on the bus when the UIC becomes master. If both queues are non-empty, the command in the high priority queue is issued first. Provided the command is not aborted, the queue issuing the command is emptied. Since the UIC becomes bus master at nondeterministic intervals, the delay between the time a message arrives in the queue and is issued on the bus is arbitrary. This nondeterminism covers two abstractions made in the model. First, it allows for any amount of latency in the link level protocol, which is not modeled. Second, it allows the time to issue an arbitrary number of messages relating to other memory blocks that may be queued ahead of the one message that is modeled.

```
MODULE receiver
VAR
  hiq : {none,response,write-shared,write-resp-shared,
         write-invalid,write-resp-invalid,invalidate};
  loq : {none,read-owned,read-shared,cty-read};
```

```
ASSIGN
  cmd :=
    case
      master & !(hiq = none) : hiq;
      master & !(loq = none) : loq;
      1 : idle;
    esac;
  init(hiq) := none;
  next(hiq) :=
    case
      !master | abort : hiq;
      1 : none;
    esac;
  init(loq) := none;
  next(loq) :=
    case
      !master | abort | !(hiq = none) : loq;
      1 : none;
    esac;
```

The local UIC can send commands to the link in response to commands observed on the local bus. Whenever a read command is sent to the link, it is entered in the remote UIC's low priority queue. If any other command is sent to the link, it is entered in the remote UIC's high priority queue. If the remote queue is full, the local bus cycle is stalled.

```
MODULE sender
DEFINE
  lopri := sending in {read-shared,read-owned,cty-read};
  hipri := sending in {invalidate,response,write-shared,
        write-invalid,write-resp-shared,write-resp-invalid};
ASSIGN
  next(remote.hiq) :=
    case
      !abort & remote.hiq = none & hipri : sending;
      1 : remote.hiq;
    esac;
  next(remote.loq) :=
    case
      !abort & remote.loq = none & lopri : sending;
      1 : remote.loq;
    esac;
  reply-stall :=
    (hipri & !(remote.hiq = none) |
     lopri & !(remote.loq = none)) union 1;
```

The local UIC sends command messages to the link in two cases. The first is to invalidate or call back cache blocks in remote caches. This occurs when the UIC is a slave and a read-owned or read-shared is received on the bus. If the UIC is in the owned state, the read-owned or read-shared is forwarded to the link. This causes the remote cache in the owned state to issue a write-resp-invalid or write-resp-shared, returning the cache block to the local bus. If the UIC is in the shared state, and a read-owned is received on the bus, an invalidate command is forwarded to the link. This causes all remote caches to go to the invalid state. Note that this may allow a processor on the local bus to write before the invalidate command has reached all remote caches. This is a possible violation of strict consistency, which is tolerated for performance reasons. Hence, the protocol does not implement a strongly consistent memory model. The memory model which the protocol does support will be discussed in more detail in the next section.

The second case in which the local UIC sends a command to the link is when the UIC has issued a read-shared or read-owned and is waiting for a response. In this case, if the UIC is a slave and a response, write-resp-shared, or write-resp-invalidate is asserted on the bus, a response is sent to the link.

```
MODULE local-UIC(remote,CMD,REPLY-OWNED,REPLY-WAITING,
                 REPLY-STALL,DATA)
ISA bus-device
ISA cache-device
ISA receiver
ISA sender
```

```
DEFINE
  sending :=
    case
      master : none;
      CMD = read-shared & state = owned : read-shared;
      CMD = read-owned  & state = owned : read-owned;
      CMD = read-owned & state = shared : invalidate;
      CMD = write-resp-invalid & waiting : write-resp-invalid;
      CMD = write-resp-shared  & waiting : write-resp-shared;
      CMD = response           & waiting : response;
      1 : none;
    esac;
```

4.1.3 The Remote UIC interface

When the UIC is remote, it behaves as if it were a main storage device. It accepts read-shared, read-owned, write-shared, and write-invalid commands from the bus, and forwards them to the local UIC via the link. When the response arrives in the high priority queue, it issues the response on the bus. In addition, it can provide a special service to caches on the local side. If the remote UIC issues a read-shared or read-owned command, and there is no reply on the remote bus (*ie.*, no slave asserts reply-owned), it is assumed that the block was copied back to main storage while the read command was in transit. The remote UIC therefore sends the read command back to the local side. This operation is called a *courtesy read*. The courtesy read will cause the main store on the local bus to respond to the original requester.

```
MODULE remote-UIC(remote,CMD,REPLY-OWNED,REPLY-WAITING,
                  REPLY-STALL,DATA)
ISA bus-device
ISA receiver
ISA sender

DEFINE
  sending :=
  case
  master :
    case
    CMD = read-shared & !REPLY-OWNED : cty-read;
    CMD = read-owned & !REPLY-OWNED : cty-read;
    1 : none;
    esac;
  !master :
    case
    CMD = read-shared & !REPLY-OWNED : read-shared;
    CMD = read-owned  & !REPLY-OWNED : read-owned;
    CMD = write-resp-invalid & waiting : write-resp-invalid;
    CMD = write-resp-shared & waiting : write-resp-shared;
    CMD = write-resp-shared & !waiting : write-shared;
    CMD = write-shared  : write-shared;
    CMD = write-invalid  : write-invalid;
    1 : none;
    esac;
  esac;
  reply-owned := 0;
```

The text for the complete model in the SMV language includes such details as an abstracted model of main storage and the cluster bus, which ties the above modules together. These are omitted here. Each cluster is modeled as an asynchronous process. Hence, the early quantification method for disjunctive relations can be used to avoid constructing the global transition relation (cf. section 2.4.2).

4.1.4 Protocol example

As an example of the protocol in operation, consider the sequence of events depicted in figures 4.2 and 4.3. In the figures, clusters 1 and 2 are both remote (*ie.*, the memory block in question resides in some other cluster). The sequence begins when a read miss occurs in a processor in cluster 2, while a processor in cluster 1 is in the owned state. At this point, the following sequence of events might occur:

1. The processor in cluster 2 issues a read-shared command on the bus, and sets its waiting flag.

2. The UIC in cluster 2 sends the read-shared command up the link, storing it in the low priority queue of the global bus UIC for cluster 2.

3. The global bus UIC for cluster 2 issues the read-shared command on the global bus, entering the shared state, and setting its waiting flag.

4. Since the global bus UIC for cluster 1 is in the owned state, it asserts reply-owned, sends the read-shared command down the link to cluster 1, enters the shared state, and sets its snoop flag.

5. The UIC in cluster 1 issues this read-shared command, entering the shared state and setting its waiting flag.

6. The processor in cluster 1 in the owned state asserts reply-owned, enters the shared state, and sets its snoop flag.

7. The processor in cluster 1 issues a write-resp-shared command, containing the block data, and clears its snoop flag.

8. The UIC in cluster 1 sends the write-resp-shared command up the link, storing it in the high priority queue of of the global bus UIC for cluster 1, and clears its waiting flag.

9. The global bus UIC for cluster 1 issues the write-response-shared command on the global bus, and clears its waiting flag.

10. (a) The global bus UIC connected to main memory sends a write-shared command containing the block data and (b) The global bus UIC for cluster 2 sends a response command, clearing its waiting flag.

11. The UIC in cluster 2 issues the response command.

12. The requesting processor in cluster 2 stores the data in its cache, and clears its waiting flag.

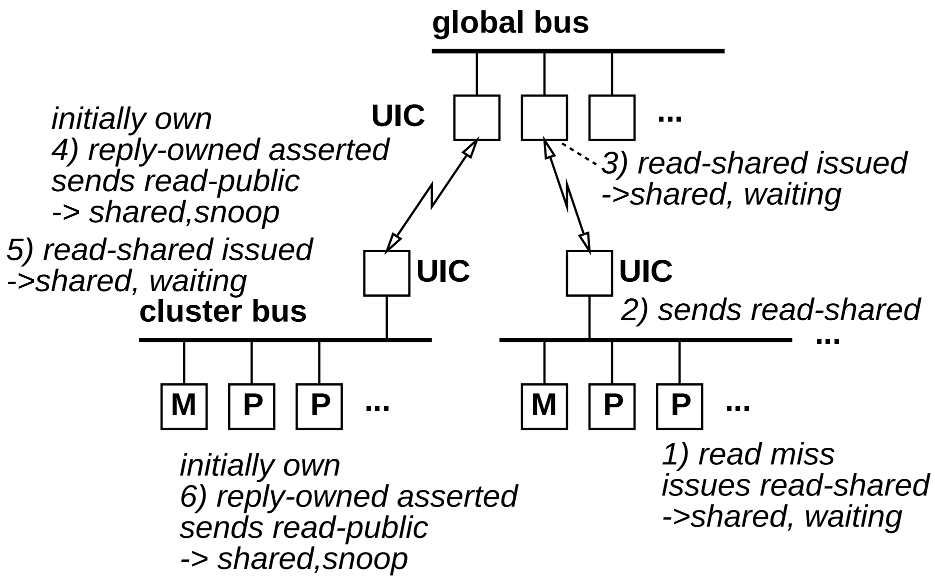


Figure 4.2: Protocol example

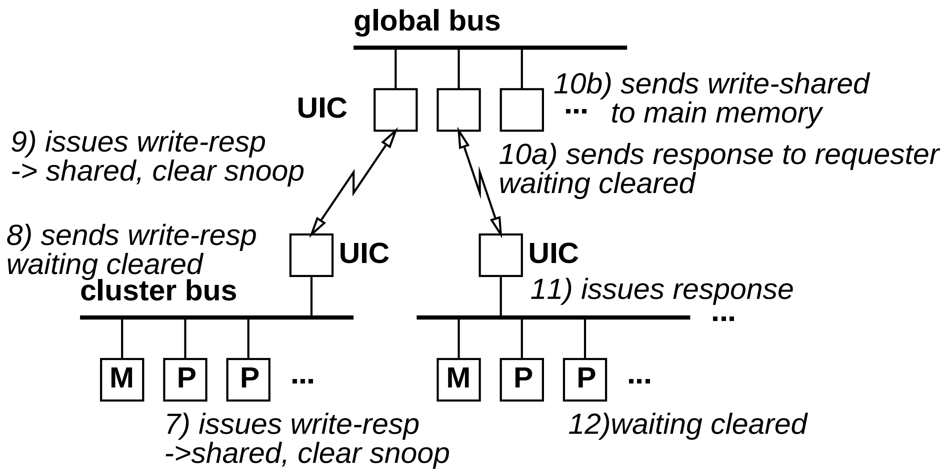


Figure 4.3: Protocol example (cont.)

4.2 Verifying the protocol

We now consider the problem of formal specification and verification of the protocol. The properties we will be concerned with are:

1. freedom from deadlock,

2. sequential consistency, and

3. local safety conditions, related to diagnostics.

Using the symbolic model checking technique, we can verify these properties automatically, despite the very large state space of the model. In fact, the model checker discovered a fairly subtle bug in the protocol – an execution sequence leading to a deadlocked state.

4.2.1 Freedom from deadlock

We will say that the protocol is deadlocked if it reaches a state in which some processor is permanently blocked from receiving access to the given memory block. Thus, our definition of deadlock takes in situations that might also be called livelock, in which the system continues to loop infinitely, but without the possibility of making progress. We can express this property in CTL with the following formula, which must hold for all processors:

$$AG(EF\mathit{readable} \wedge EF\mathit{writable}) \tag{4.1}$$

In other words, it it always possible that the memory block will become readable by the given processor, and always possible that it will become writable. We can check this property using SMV by adding the following specification to the processor module:

```
SPEC
  AG(EF readable & EF writable)
```

The specification turns out to be false, and as a counterexample, the model checker produces an execution trace leading to a deadlocked state. This is an actual bug in the original protocol which was found by the model checker, but not in behavioral simulations. The complexity of the counterexample, and the unusual sequence of events that leads to the deadlock should give some indication of why this error would be unlikely to occur in random simulations. The time required to produce the counterexample was slightly under ten minutes running on a Sun 3/60.

The steps of the counterexample are depicted in figures 4.4 to 4.6. Cluster 1 is the local cluster, and clusters 2 and above are remote clusters. We pick up the counterexample at a point where a processor in cluster 2 is in the owned state:

1. A read miss occurs in a processor in cluster 1. This processor issues a read-shared command on the bus. It enters the shared state and sets its waiting flag.

2. Since the UIC in cluster 1 is in the owned state, it asserts reply-owned, enters the shared state, and sends a read-shared command up the link, storing it in the low priority queue of the global bus UIC for cluster 1.

3. A processor in cluster 3 also issues a read-shared command. As a result, the global bus UIC for cluster 3 issues the read-shared command on the global bus, entering the shared state, and setting its waiting flag.

4. Since the global bus UIC for cluster 2 is in the owned state, it asserts reply-owned, sends a read-shared command down the link to cluster 2, enters the shared state, and sets its snoop flag.

5. The UIC in cluster 2 issues this read-shared command, entering the shared state and setting its waiting flag.

6. The processor in cluster 2 in the owned state asserts reply-owned, enters the shared state, and sets its snoop flag.

7. The processor in cluster 2 issues a write-resp-shared command, containing the block data, and clears its snoop flag.

8. The UIC in cluster 2 sends the write-resp-shared command up the link, storing it in the high priority queue of of the global bus UIC for cluster 1, and clears its waiting flag.

9. The global bus UIC for cluster 2 issues the write-response-shared command on the global bus, and clears its waiting flag.

10. (a) The global bus UIC connected to main memory (cluster 1) sends a write-shared command containing the block data and (b) The global bus UIC for cluster 3 sends a response command, clearing its waiting flag.

11. The UIC in cluster 1 issues the write-shared command.

12. The block data are stored in main memory.

13. A processor in cluster 3 again issues a read-shared command. As a result, the global bus UIC for cluster 3 issues the read-shared command on the global bus, entering the shared state, and setting its waiting flag.

14. Since read-owned is not asserted, the UIC for cluster 1 sends the read-shared command down the link towards main memory.

At this point, the system is deadlocked. The original read-shared command sent in step 1 in cluster 1 is still in the low priority queue at th global bus level, but is stalled by the waiting flag set in the global UIC for cluster 3. Similarly, the read-shared command sent by cluster 3 is in the low priority queue in the cluster 1 UIC, but is stalled by the waiting flag of the original requester. This is an example of the classic deadlock situation which occurs when two processes attempt to obtain locks on two resources (in this case two buses) in different orders. Nonetheless, the sequence of events that lead to this situation were sufficiently complex that the designers did not anticipate that the situation could occur, and simulations did not produce it. In fact, the deadlock situation was found at a search depth of thirteen transitions. At each step in this sequence, there were several alternatives that might have averted the deadlock. Thus it is possible, but unlikely that this

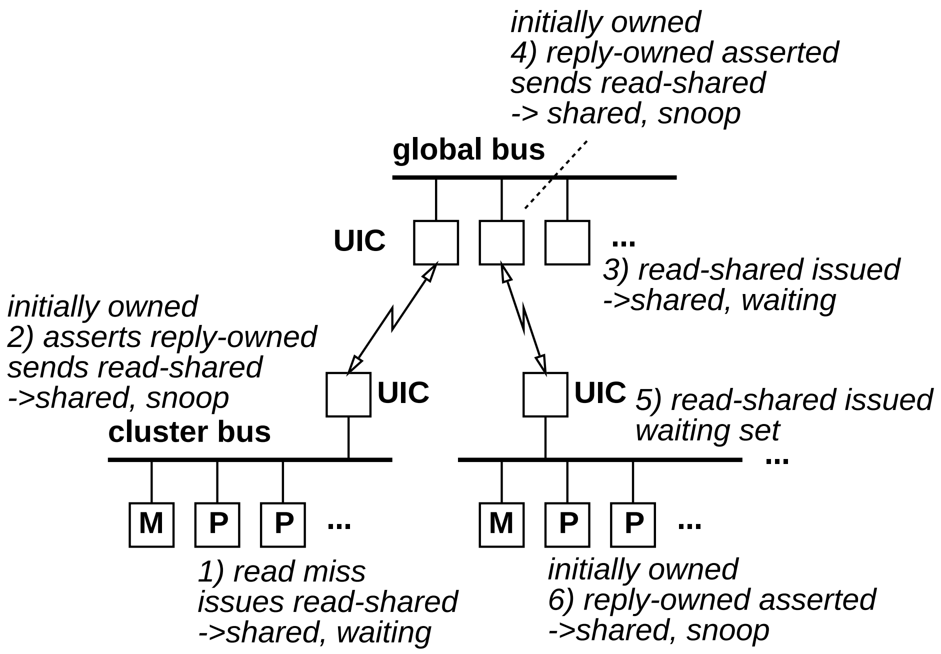


Figure 4.4: Deadlock example

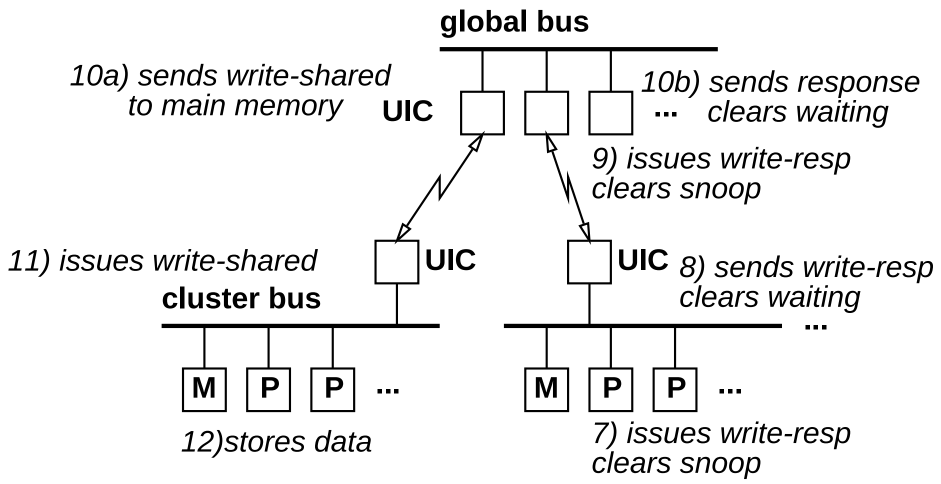


Figure 4.5: Deadlock example (cont.)

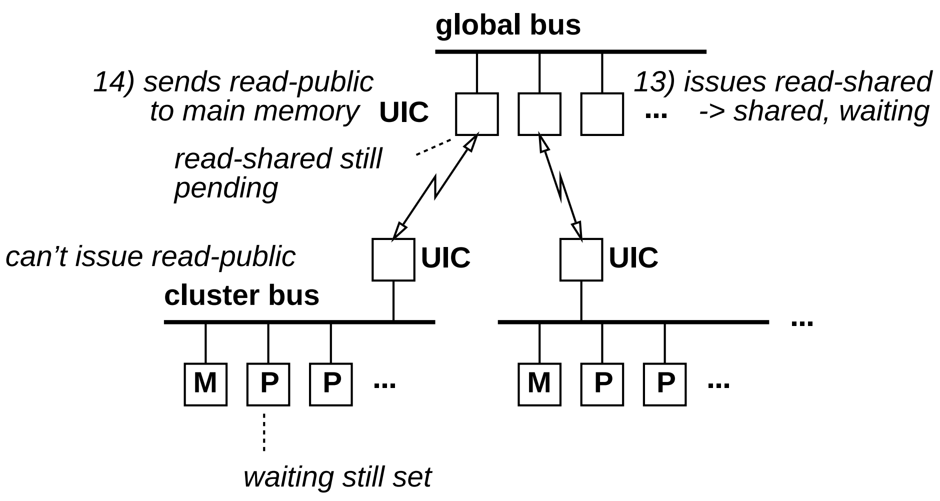


Figure 4.6: Deadlock example (cont.)

deadlock would be found by a random simulation run, or a simulation run based on address traces.[1]

The fact the the model checker was able to print out automatically an example of this deadlock highlights an important practical aspect of the technique. Counterexamples are of perhaps even greater value than a proof that the system is correct, since such a proof is based on the assumption that the system is correctly modeled, and the specification is correct and complete. A counterexample, however, provides an important clue as to where a bug in the system lies, and how it might be corrected.

4.2.2 Correcting the deadlock

The problem causing the deadlock is that the remote owner of the memory block can write the data back to main memory while a read

[1]In fact, the number of possible transitions from a given state ranges from 6 to 12. The probability of a random simulation run executing this trace is therefore in the range $6^{-13} = 7.7 \times 10^{-11}$ to $12^{-13} = 9.3 \times 10^{-15}$. The expected time for a random simulation to exhibit this behavior would be somewhere between 2.4 years and 29 millenia, assuming the simulation could be carried out at 10,000 steps per second.

command from the local cluster is in transit to the remote cluster. The write command crosses the read command in the mail, so to speak. A remote request for the same block can then lock the global bus, leading to deadlock. The Encore engineers corrected the deadlock problem in the following way. The write command, when it reaches the local bus, is converted by the UIC into a write-response command. This supplies data to the local requester and frees the local bus. Unfortunately, it also leaves an orphan read command in the system. If a read command from the remote side is issued on the local bus, and a remote processor subsequently reaches the owned state, the orphan read command will disrupt th protocol. To prevent this, when the orphan read is issued, it is converted to a special command called *echo-response*, which is sent back to the local cluster. The UIC in the local cluster stalls any commands on the local bus until the echo-response arrives, thus guaranteeing that the orphan read command is destroyed.

The corrected model satisfies the absence of deadlock specification. The performance of the SMV model checker in verifying this is plotted in figure 4.7, for a model with 2 clusters, as the number of caches in each cluster is increased from 2 to 6 (thus, in the largest model, there are 12 caches and 4 UICs). Part (a) shows the run time as a function of the number of caches per cluster. Part (b) shows the number of OBDD nodes used overall, and for representing the transition relation. Part (c) shows the number of reachable states of the model. Although the run time points are well fit by a quadratic curve, the actual asymptotic performance is most likely cubic, as in the case of the synchronous arbiter (cf. section 2.4.1), owing to linear increases in the transition relation size, the number of fixed point iterations and the size of the OBDDs representing fixed point approximations.

Since the number of bus wires running between successive caches is fixed, we can apply theorem 7 to show that the transition relation OBDD size must grow linearly in the number of caches. The fact that the fixed point approximation OBDDs also grow linearly bears further examination, however. This phenomenon can be understood by considering the nature of the protocol. Imagine cutting a cluster bus in half, and consider how much information must be communicated from one half of the bus to determine whether a given state of the system is in the reachable set or not. In fact, this amount is fixed, independent of the number of caches on the bus, since we need only know if there are any caches in the shared state or the owned state on the other side of the cut, and not in particular which caches these are or how many. As a result, the number of OBDD nodes (representing the reached state set) at the level corresponding to our cut is bounded.[2] This is characteristic of bus snooping protocols, and other protocols which are "loosely coupled", in the sense that one half of the system has bounded knowledge of the state of the other half of the system.

As part (c) of the figure shows, the number of states of the system increases exponentially with the number of caches per cluster. Despite this, the performance of the symbolic model checking algorithm is polynomial. Thus, for this particular model and specification, we have solved the state explosion problem.

4.2.3 Sequential consistency

When writing a formal specification for the Gigamax cache consistency protocol, we need to consider the model of a distributed memory which the Gigamax provides to the programmer. As mentioned previously, for performance reasons the protocol does not maintain strict consistency of the caches. A cache block in the shared state may be out of date for a short time while an invalidate message is traversing the network. This is tolerated, since maintaining strict consistency would require an acknowledgment of invalidation to be collected from all caches in the shared state before a cache block could be modified.

There are a number of distributed memory models that may be supported by such a system. A *totally ordered* model is one in which all processors observe all values written to the memory in the same order. For example, in a totally ordered model, if the processors write into a location the sequence of values $1, 2, 3, \ldots$, then all processors which read the location will observe any new values to be greater than or equal to all previous values. We will show that the Gigamax protocol has this property, for a one block system. In a partially ordered model, values written may in some cases be observed in a different order by different processors. Some guarantee of ordering is usually made. For

[2]For other applications of this kind of argument, see [Bry91].

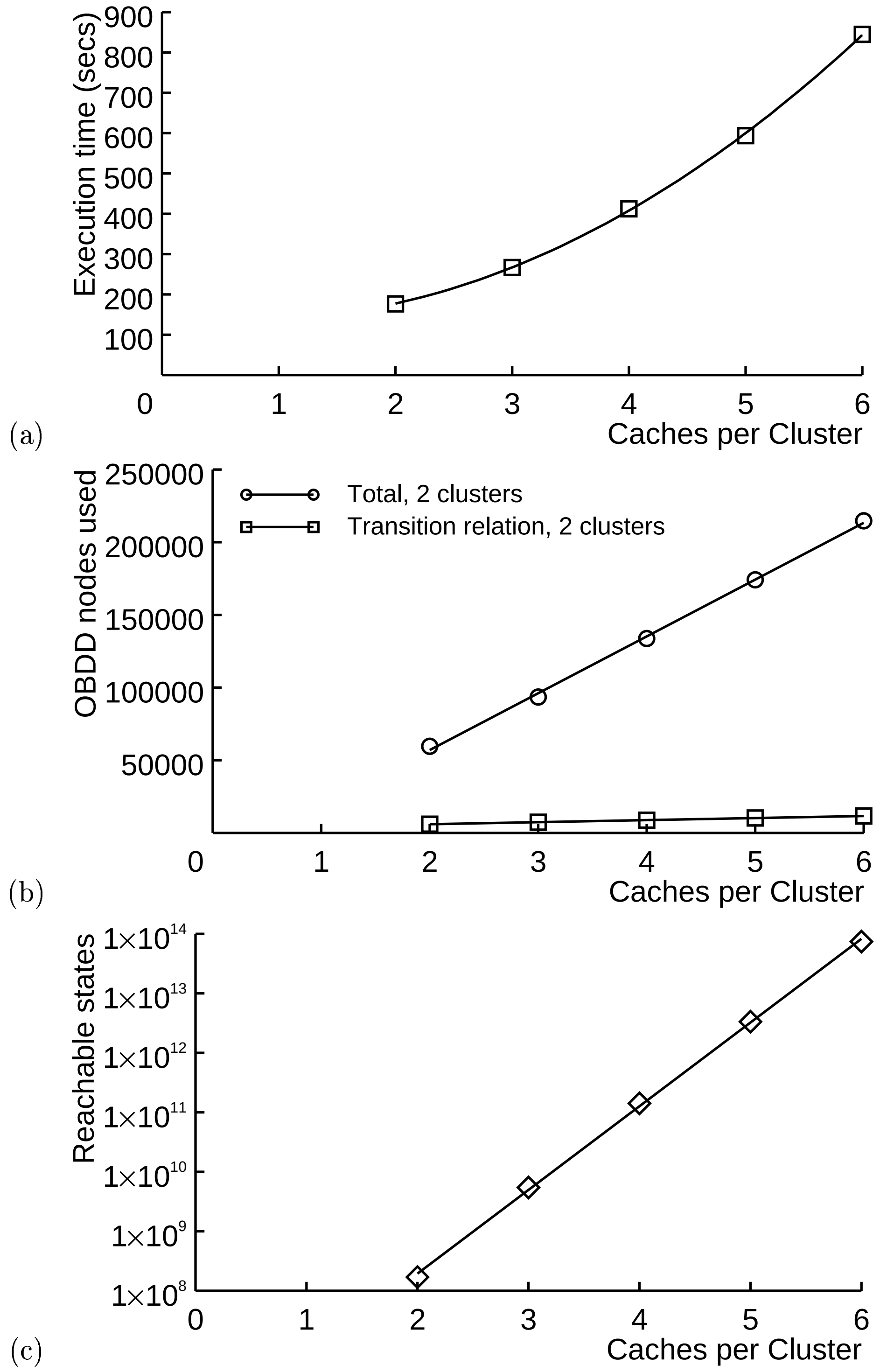


Figure 4.7: Performance for checking deadlock

example, all writes must be observed in the same order relative to special synchronization operations, or all writes by the same processor must be observed in the same order. The latter model is supported by the Gigamax protocol for writes to different cache blocks. Since our model of the protocol only describes the behavior one cache block, however, the model cannot be used to check this property.

Returning to the problem of total ordering of writes to the same block, it might seem at first that there is no "finite state" description of a protocol that writes an unbounded sequence of values. We can check the property, however, by using an abstraction. We do this by choosing a value n, and storing in the model only one bit of information – whether the data value is less than n or greater than or equal to n. We then assume that the processors never write a value less than n after a value greater than n has been written, and we show that a processor never reads a value less than n after reading a value greater than n. Since the value of n is arbitrary, it follows that all processors read data values in non-decreasing order, satisfying the total ordering requirement. We now consider how to model the system using this abstraction. For each cache, we introduce a variable whose value is 0 when the data value is less than n and 1 when the date value is greater than or equal to n. This variable may change whenever the block is writable, but may only change from 0 to 1, since we assume the processors only increase the data value. The following SMV code models the data held in the processor's cache:

```
MODULE data-device
VAR
  data : boolean;
ASSIGN
  next(data) :=
    case
      !master & waiting & CMD in {response,write-resp-invalid,
                       write-resp-shared} : DATA;
      writable : data union 1;
      1 : data;
    esac;
DEFINE
  data-enable := master & CMD in {response,write-resp-invalid,
                    write-resp-shared,write-invalid};
```

Additionally, we introduce variables to represent the values on the buses and the values in the high priority message queues. The low priority queues hold only requests, which have no data value.

We would now like to prove that this abstract model of the data path of the protocol satisfies the following specification in CTL, for all processors:

$$AG[(\mathit{readable} \wedge \mathit{data} \geq n) \Rightarrow AG\neg(\mathit{readable} \wedge \mathit{data} < n)] \tag{4.2}$$

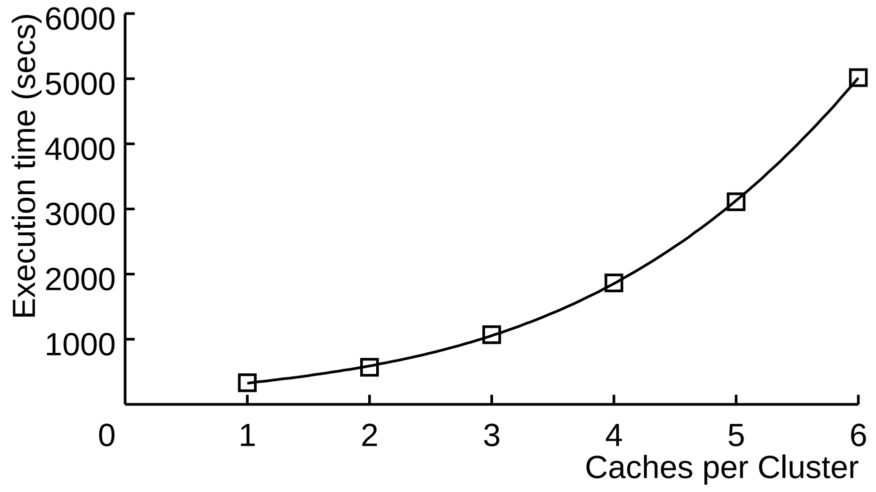
In other words, if ever a value greater or equal to n is observed, a value less than n is never observed in the future. We can check this using SMV by adding the following specification to the processor module:

```
SPEC
  AG(readable & data -> AG (readable -> data))
```

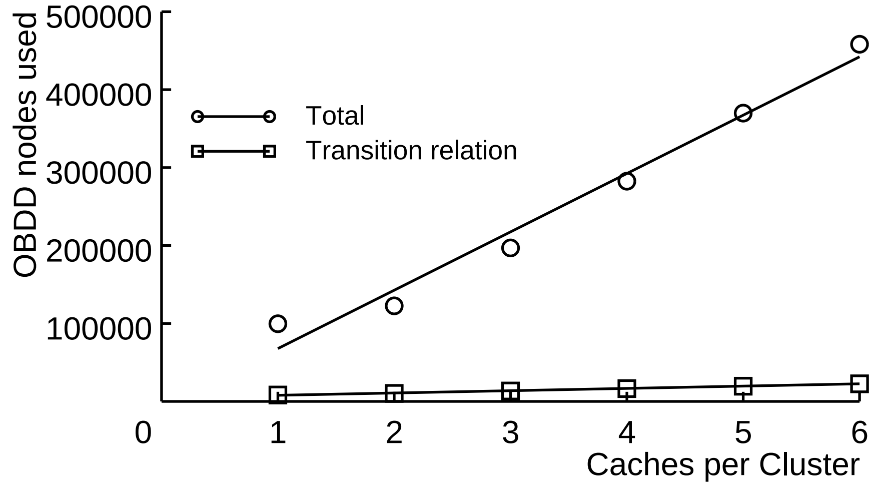
Figure 4.8 shows the performance of the symbolic model checking algorithm in verifying this formula, again for a model with 2 clusters. Part (a) of the figure shows the execution time, while part (b) shows the amount of storage used. Notice that although the execution times are roughly ten times those obtained for the model without data, they are still cubic in the number of processors per cluster.

4.2.4 Correctness of diagnostics

In addition to the above specifications, it was also particularly useful to check that the diagnostics built into the protocol never flagged an error under normal operation of the protocol. Errors are flagged by the diagnostic system in each processor subsystem whenever a command is observed on the bus which is inconsistent with the processor's local state. Determining which command/state combinations are normal, and which are errors is difficult, and a number of errors of this type were found in the protocol using the model checking technique.



(a)



(b)

Figure 4.8: Performance for checking sequential consistency

4.3 Evaluation

In verifying the Gigamax model with respect to the formal specifications, the symbolic model checker was able to perform an exhaustive search of the model's state space without explicitly constructing the global state graph. As a result, the state explosion problem was avoided. In addition, the model checker exposed a number of subtle errors in the design that were not found in simulation. These errors were usually caused by events (*eg.*, cache misses and message arrivals) occurring out of the normal sequence anticipated by the designers. This type of error is difficult to find in random simulations, since the probability of a given sequence of random events occurring by pure chance is in inverse exponential proportion to the length of the sequence. As we have seen, the sequences necessary to produce protocol errors can be quite long. As the design evolved to correct the errors found by model checking, the model was easily adapted, and quickly provided an analysis of any new errors introduced by design changes. This tends to amortize the initial effort required to produce the protocol model. The ability of the symbolic model checker to find errors quickly makes it easier to experiment with alternative designs, and also helps to build the designer's intuition about the behavior of the system. This is important, because designers tend to concentrate on normal sequences of events, and overlook the unusual sequences. The use of OBDDs in the symbolic model checker made it possible to check a model that would have been very time consuming, or perhaps impossible to check using earlier algorithms.

At this point, the technique has a number of limitations. One limitation is the use of OBDDs. For example, while we find the OBDD sizes growing polynomially in the number of caches in the Gigamax model, if we instead increase the number of cache blocks and leave the number of caches constant, we find the size of the OBDDs increasing exponentially. As a result, it was extremely difficult to check specifications of a system with just two cache blocks (some runs took up to a week, and others never finished). In these cases, the size of the OBDDs representing the fixed point approximations became intractably large. When this happens, techniques such as early quantification that make the representation of the transition relation smaller are little use, since they do not effect the size of the OBDDs representing fixed point approximations.

Another major issue is implementation of the protocol. Clearly, verification of the protocol itself is important, since a correct protocol is a prerequisite for a correct implementation. This is, of course, only half the story. Techniques are also needed to insure that the verified protocol is implemented correctly in hardware. This can in fact be done, using a process of successive refinement of finite state systems that has been studied extensively by Kurshan [Kur87]. The work of Bose and Fisher [BF89a] is also an example of this. Unfortunately, the truth of CTL formulas containing existential quantifiers is not necessarily preserved by this kind of refinement. Thus, for example, though the high level protocol may be deadlock free, a specific implementation of the protocol may not be deadlock free[3]. In order for the implementation to preserve all CTL properties of the protocol, the two would have to be bisimular (cf. section 2.6.1). Since this is a very strong requirement, it is not clear that the protocol could in fact be implemented with this degree of accuracy in an efficient way. For essentially this reason, Grumberg and Long have studied the use of a subset of CTL using only universal path quantifiers for hierarchical reasoning [GL91]. In any event, though checking the absence of deadlock specification was very useful in finding bugs in the protocol, we must attach a special caveat to this result, since it does not guarantee that all reasonable implementations of protocol will be deadlock free.

Finally, there is the problem of verifying a model with a finite number of processors, when there is no finite limit on the number of processors that could in principle be added to the system. In practice, the intended maximum number of processors is approximately 100. Even using the symbolic model checking technique, however, checking a system of 100 processors seems infeasible at present, and 1000 processors is out of the question. To deal with systems with a very large num-

[3]In fact, such a deadlock, involving an interaction between the memory and processor subsystems was known to the Encore engineers. The memory system, when busy, would stall any new requests, but the stalled request would still remain in the memory system's pipeline for four clock cycles. Thus, when the processor retried the request four clock cycles later, it would be stalled again, and the process would repeat indefinitely.

ber of identical components, we can apply methods of induction over processes. As in the case of successive refinement, induction methods are not fully automatic – some human input is required in the form of an inductive hypothesis. In the next chapter, we will deal with the problem of induction over processes.

Chapter 5

Induction and model checking

This chapter deals with the verification of systems that have an arbitrary number of similar components, arranged in some inductively defined structure. Systems of this type are commonplace – they occur in bus protocols and network protocols, I/O channels, and many other structures that are designed to be extensible by adding similar components. After using a model checking system to determine the correctness of a system configured with a fixed number of processors or other components, it is natural to ask whether this number is enough in some sense to represent a system with any number of components. For example, a Gigamax system can be built by connecting some arbitrary number of cluster buses to a global bus, then filling each cluster bus with an arbitrary number of processor cards. It is practically impossible to verify using model checking methods alone that all possible configurations of the system satisfy the specifications, even given a physical bound on the number of cards in a backplane. However, by supplying an appropriate inductive hypothesis, we can in many cases reduce the problem of verifying a system of arbitrary size to one of verifying a system of fixed size. The inductive hypothesis can take the form of a finite state process.

5.1 The general framework

Induction over systems of processes can be put in a fairly general framework, which is independent of the mechanics of the process model, relying only on certain algebraic properties of the operators for combining processes. Let us assume that we have a collection of processes, and a collection of operators acting on processes. In a typical process model, we have some form of parallel composition operator, some form of operator for renaming signals, and perhaps a hiding operator, which makes a given signal invisible to the outside. The exact choice of operators is not material here, however. We require only that the operators be monotonic with respect to a reflexive transitive relation $\leq$ on processes. The idea of this order is that if $p \leq q$, the p is in some sense more specific, or more deterministic, than q. The properties we wish to verify should be preserved as we descend the order.

As an example of induction on processes, suppose we have a parallel composition operator $\|$ on processes, which is monotonic with respect to a pre-order $\leq$. In this case, we can apply the following induction rule:

$$\frac{\begin{array}{c} p \leq q \\ q \parallel p \leq q \end{array}}{p \parallel \cdots \parallel p \leq q}$$

Think of the inequalities $p \leq q$ and $q \parallel p \leq q$ as substitution rules. If $p \leq q$, we can safely substitute p for any occurrence of q in a given term, in the sense that we will only make the term lesser in the partial order. Thus, we can always substitute p for q on the lesser side of an inequality. For example, if $q \parallel p \leq q$, we have

$$\begin{array}{rcl} q \parallel p & \leq & q \\ (q \parallel p) \parallel p & \leq & q \\ ((q \parallel p) \parallel p) \parallel p & \leq & q \\ & \cdots & \end{array}$$

If $p \leq q$, we can substitute p for q, giving us $p \parallel \cdots \parallel p \leq q$. We call q a *process invariant.* Other induction rules can be generated, based on other substitutions. For example, assume we have a parallel

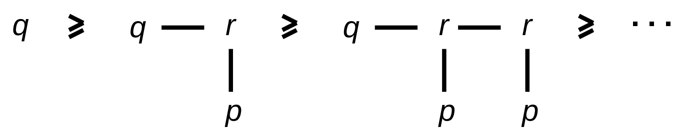


Figure 5.1: Processes generated by safe substition

composition operator $\|$ and a renaming operator ϕ, both monotonic with respect to $\leq$. Then we have

$$\frac{\phi(q) \parallel r \parallel p \leq q}{\phi(\phi(\,\cdots\,) \parallel r \parallel p) \parallel r \parallel p \leq q}$$

Given a collection of substitution rules, we can inductively generate a class of processes from any process q. For example, figure 5.1 depicts the first few processes in the class generated by the above induction rule. Every process in the class is smaller than q in the partial order. Thus, any properties of q which are preserved as we descend the partial order are inherited by all the processes in the class. The key, then, is to choose a partial order that preserves the properties we are interested in verifying. The most straightforward way to do this is to choose a class of properties that we wish to preserve, and then define the partial order accordingly.

For example, suppose we wish to preserve all properties expressible in the logic CTL. In this case, the partial order we obtain is a degenerate one, which partitions the Kripke models into a set of incomparable equivalence classes. To see this, assume towards a contradiction that p satisfies every CTL formula satisfied by q, and there is some formula f satisfied by p but not by q. In this case, it follows that q satisfies $\neg f$. This implies, however, that p satisfies $\neg f$, a contradiction. Hence, if $p \leq q$, then p and q satisfy the same set of CTL formulas. Since CTL characterizes Kripke models up to bisimulation [BCG87], it follows that p and q are bisimular. This is unfortunate, since we do not want our induction framework to apply only to classes of Kripke structures that are equivalent. In general, we would like to treat systems whose behavior becomes more specific as we add processes to the system.

One way to do this is to use a subset of the logic. For example, suppose we choose to preserve those formulas which use only universal path quantifiers. This subset is called ∀-CTL [GL91]. A formula in CTL is also in ∀-CTL if driving the negations in to the literals results in a formula without the E path quantifier. Examples of ∀-CTL formulas are

$$\begin{aligned} AG\neg EGp &\equiv AGAF\neg p \\ \neg EGEXp &\equiv AFAX\neg p \end{aligned}$$

Examples of CTL formulas which are not in ∀-CTL are

$$\begin{aligned} AG\neg AFp &\equiv AGEG\neg p \\ \neg EGAXp &\equiv AFEX\neg p \end{aligned}$$

Clearly, if a formula f contains path quantifiers, then f and $\neg f$ cannot both be in ∀-CTL. Grumberg and Long [GL91] have shown that if p satisfies every ∀-CTL formula satisfied by q, then q simulates p, and conversely. Simulation is easily shown to be reflexive and transitive. Thus simulation is a pre-order suitable for inductive proofs of ∀-CTL formulas. Let $p \leq q$ iff q simulates p. This gives us the following induction rule:

$$\frac{\begin{array}{c} q \models f \quad (f \in \forall\text{-CTL}) \\ p \leq q \\ q \parallel p \leq q \end{array}}{p \parallel \cdots \parallel p \models f}$$

as well as other rules engendered by various systems of safe substitutions. Recall from section 2.6.1 that simulation is the greatest relation between the states of q and the states of p such that if x simulates y, then:

1. x and y agree on the atomic propositions, and
2. every successor of y is simulated by a successor of x.

A Kripke model q simulates p if every initial state of p is simulated by some initial state of q. Since this relation can be expressed as a greatest fixed point in the Mu-Calculus, it can be verified automatically using the symbolic model checking technique. The fact that simulation is not symmetric allows us more flexibility in constructing systems using substitution rules than we would have using bisimulation.

5.1.1 Induction in other models

We can set up an induction framework for a variety of models by establishing a pre-order and a set of monotonic process operators. For models of concurrent automata (such as the *s/r* model [Kur85]), the natural partial order is language containment.

In the *s/r* model, a process is an automaton which accepts infinite strings over a Boolean algebra. There are two natural operators over this class of processes. The automaton product operation simulates parallel execution, while Boolean algebra homomorphisms can be used to induce a renaming or abstraction of the variables by which processes communicate. Kurshan shows that both of these operations respect the relation of language containment between automata [Kur86]. An example of induction in this framework can be found in [KM89].

Induction can also be applied in process algebras like CCS [Mil80] which are based on two-way synchronization. In this case, there is a variety of plausible process relations, including observational equivalence, weak observational equivalence, and a number of pre-order relations on processes. An induction example using the "may" pre-order for CCS processes can also be found in [KM89].

5.2 Induction and SMV

An induction framework can be set up for the SMV.0 language, based on either simulation or bisimulation. This framework includes two kinds of process operators – the parallel composition operator $||$ and renamings operators based on maps ϕ from locations to locations. We will show that both are monotonic with respect to simulation and bisimulation.

5.2.1 Proving compositionality

Recall that semantically, an SMV.0 program denotes a triple (T, I, R), where T assigns types to locations, I is the set of initial states, and R is the transition relation. There are two basic process operators provided by SMV: instantiation and parallel composition. An instantiation results from a map ϕ on locations (a renaming). This renaming induces a map Φ from states of $\phi(M)$ to states of M, such that for all locations l, $\Phi(x)(l) = x(\phi(l))$. If $M = (T, I, R)$ is a process, then $\phi(M) = (T', I', R')$ where

1. $T'(\phi(l)) = T(l)$,
2. $I' = \{x \mid \Phi(x) \in I\}$ and
3. $R' = \{(x, y) \mid (\Phi(x), \Phi(y)) \in R\}$.

The rules for renaming require that no hidden location can be renamed to a visible location, and no two distinct locations can be renamed to the same hidden location. The following lemma and theorem show that this definition of renaming is a suitable operation for inductive reasoning using simulation or bisimulation:

Lemma 4 *Let ϕ be a legal renaming, and let Φ be the state map induced by ϕ. If $x'_1 = \Phi(x_1)$ and x'_1 agrees with x'_2 on the visible locations, then there exists x_2 which agrees with x_1 on the visible locations, such that $x'_2 = \Phi(x_2)$.*

Proof. Construct x_2 as follows: For every location l, if l is in the range of ϕ, then choose any l' such that $l = \phi(l')$, and let $x_2(l) = x'_2(l')$. Otherwise, let $x_2(l) = x_1(l)$.

First we show that $x'_2 = \Phi(x_2)$. For all locations l'', if $\phi(l'')$ is visible, then l'' must also be visible (since hidden locations cannot legally be renamed to invisible locations). Let $l = \phi(l'')$. Since l is in the range of ϕ, there is some visible l' such that $l = \phi(l')$ and $x_2(l) = x'_2(l')$. Since x'_1 and x'_2 agree on the visible locations, $x'_2(l') = x'_1(l')$. Since $x'_1 = \Phi(x_1)$, $x'_1(l') = x_1(l) = x'_1(l'') = x'_2(l'')$. Thus $x'_2(l'') = x_2(\phi(l''))$. On the other hand, if $\phi(l'')$ is hidden, there exists no other location l' such that $\phi(l') = \phi(l'')$, since two locations cannot legally be renamed to the same hidden location. Therefore $x_2(\phi(l'')) = x'_2(l'')$. Thus, by definition $x'_2 = \Phi(x_2)$.

Second, we show that x_1 and x_2 agree on the visible locations. Let l be any visible location. If l is not in the range of ϕ, then $x_1(l) = x_2(l)$ by construction. Otherwise, there is an l' such that $\phi(l') = l$ and $x_2(l) = x'_2(l')$. Since l' must be visible, $x'_2(l') = x'_1(l') = x_1(\phi(l')) = x_1(l)$. □

Theorem 9 *Instantiation of SMV.0 modules is monotonic with respect to simulation and bisimulation.*

Proof. Imagine that we have two processes, M_1 and M_2, such that M_2 simulates M_1.

First, let x_1 and x_2 be states of $\phi(M_1)$ and $\phi(M_2)$ respectively. We show by induction that if x_1 and x_2 agree on the value of all visible locations, and if $\Phi(x_2)$ simulates $\Phi(x_1)$, then x_2 n-simulates x_1, for all n.

The basis case is trivial, since x_1 0-simulates x_2 exactly when x_1 and x_2 agree on the value of all visible locations.

For the induction step, let (x_1, y_1) be any transition of $\phi(M_1)$. By definition, $(\Phi(x_1), \Phi(y_1))$ is a transition of M_1. Hence, there exists a transition $(\Phi(x_2), y'_2)$ of M_2 such that $\Phi(y_1)$ simulates y'_2. Since $\Phi(y_1)$ simulates y'_2, $\Phi(y_1)$ and y'_2 agree on the visible locations. By the lemma, there must exist y_2 which agrees with y_1 on the visible locations, such that $y'_2 = \Phi(y_2)$. By inductive hypothesis, y_2 $(n-1)$-simulates y_1, therefore x_2 n-simulates x_1.

Now we show that every initial state of $\phi(M_1)$ is simulated by some initial state of $\phi(M_2)$. A state x_1 is initial in $\phi(M_1)$ exactly when $\Phi(x_1)$ is initial in M_1. Let $x'_1 = \Phi(x_1)$. If x'_1 is initial in M_1, then it is simulated by some x'_2 which is initial in M_2. Since x'_1 is simulated by x'_2, they agree on the values of the visible locations. Hence, by the lemma, there exists x_2 which agrees with x_1 on the visible locations, such that $x'_2 = \Phi(x_2)$. By the above argument, x_1 is simulated by x_2. Therefore, $\phi(M_1)$ is simulated by $\phi(M_2)$.

We can prove that renaming respects bisimulation by the same argument, applied symmetrically. □

The parallel composition of SMV.0 programs is formed by taking the union of the type functions T and the intersections of the initial sets I and the transition relations R, after renaming the hidden variables of each process onto disjoint spaces. We can show that this operation is also suitable for inductive reasoning:

Theorem 10 *Parallel composition of SMV.0 programs is monotonic with respect to simulation and bisimulation.*

Proof. Imagine that we have four processes, M_1, M_2, M'_1 and M'_2, such that M_1 is simulated by M_2 and M'_1 is simulated by M'_2.

Let ϕ and ϕ' be the renamings associated with parallel composition. These are identities over the visible locations, and map the hidden locations onto disjoint ranges. Let Φ and Φ' be the induced state maps. (Thus, given a state of a parallel composition $M \parallel M'$, Φ yields the corresponding state of M and Φ' yields the corresponding state of M'.) Let x_1 be a state of $M_1 \parallel M'_1$ and let x_2 be a state of $M_2 \parallel M'_2$. We show by induction that if $\Phi(x_1)$ is simulated by $\Phi(x_2)$ and $\Phi'(x_1)$ is simulated by $\Phi'(x_2)$, then x_1 n-simulates x_2, for all n.

For the base case, since ϕ is the identity for the visible locations, $\Phi(x_1)$ agrees with x_1 on the visible locations, as does $\Phi(x_2)$ with x_2. Since $\Phi(x_1)$ is simulated by $\Phi(x_2)$, they also agree, therefore x_1 0-simulates x_2.

For the induction step, let $u_1 = \Phi(x_1)$, $u'_1 = \Phi'(x_1)$, $u_2 = \Phi(x_2)$, $u'_2 = \Phi'(x_2)$. Let (x_1, y_1) be a transition of $M_1 \parallel M'_1$, and let $v_1 = \Phi(y_1)$ and $v'_1 = \Phi'(y_1)$. By definition, (u_1, v_1) is a transition of M_1 and (u'_1, v'_1) is a transition of M'_1. Since u_1 is simulated by u_2 and u'_1 is simulated by u'_2, there must exist v_2 and v'_2 such that (u_2, v_2) is a transition of M_2, (u'_2, v'_2) is a transition of M'_2, v_1 is simulated by v_2 and v'_1 is simulated by v'_2. Now we construct y_2. For all visible locations l, let $y_2(l) = y_1(l)$. For all hidden locations l in the range of ϕ, there is a unique l' such that $l = \phi(l')$, since a renaming cannot legally map distinct locations on to the same hidden location. Let $y_2(l) = v_2(l')$. Similarly, for all hidden locations l in the range of ϕ', there is a unique l' such that $l = \phi'(l')$. Let $y_2(l) = v'_2(l')$. By this construction, $v_2 = \Phi(y_2)$ and $v'_2 = \Phi'(y_2)$. Hence, by inductive hypothesis, x_2 $(n-1)$-simulates y_2. By definition, (x_2, y_2) is a transition of $M_2 \parallel M'_2$. Therefore x_1 n-simulates x_2.

Now we show that every initial state of $M_1 \parallel M'_1$ is simulated by an initial state of $M_2 \parallel M'_2$. Let x_1 be an initial state of $M_1 \parallel M'_1$ and let $u_1 = \Phi(x_1)$, $u'_1 = \Phi'(x_1)$. By definition, u_1 is initial in M_1 and u'_1 is initial in M'_1. Hence there exist u_2 and u'_2 such that u_1 is simulated by u_2, u'_1 is simulated by u'_2, u_2 is initial in M_2 and u'_2 is initial in M'_2. We can construct x_2 such that $u_2 = \Phi(x_2)$ and $u'_2 = \Phi'(x_2)$ in the same manner as we constructed y_2 above. By the above argument, x_1 is simulated by x_2.

We can prove that parallel composition respects bisimulation by the same argument, applied symmetrically. □

5.2.2 Computing simulation relations

Since the simulation and bisimulation relations can be expressed in the Mu-Calculus (cf. section 2.6.1), they can be computed using the symbolic model checking technique. In this way, we can automatically test whether substituting a given module p for another module q is safe, in the sense of preserving all CTL or ∀-CTL properties.

There are a few techniques that can improve the efficiency of this process. The simplest is to note that simulation between two states implies that they agree on the values of the visible locations. Therefore, there is no need to use separate OBDD variables to encode the visible locations of the two processes when representing the simulation relation. As with CTL model checking, we can compute the reachable state space of the two programs, and use these sets to restrict the computation of the equivalence relation. In cases where the simulation relation cannot be computed, we can instead compute a stronger relation between the programs, which requires that all pairs of states which are simultaneously reachable (reachable along paths which agree on the visible locations) are 1-simular. This relation can be tested by a forward search of the reachable state space of the composition of the two programs. In the case of deterministic programs (in which no two successors of a given state agree on all of the visible locations), this amounts to a test of string language containment [GL91]. In either approach, if the test fails, we can extract as a counterexample a pair of paths, such that all corresponding states are 0-simular, and the last pair fails to be 1-simular. This test can be used to formulate another guess for the process invariant, until a sound invariant is found.

SMV supports induction in the following way. Each hypothesis of an induction rule is of the form `p` $\leq$ `q`, where `p` and `q` are modules. This is completely general, since module `p` can be an arbitrary parallel composition of instances of other modules. By inserting the declaration

```
SIMULATES p
```

in module `q`, we cause the SMV model checker to test whether `q` simulates `p` and if not, to produce a counterexample.

5.2.3 Induction and SMV.1

Is it possible to extend the above framework to SMV.1, which includes interleaving processes? Unfortunately, the answer is no. Consider, for example, the following two modules, which are bisimular:

```
MODULE a
VAR
  x : boolean;
ASSIGN
  init(x) := 0;
  next(x) := 0;

MODULE b
  x : boolean;
ASSIGN
  init(x) := 0;
  next(x) := x;
```

Note, however, that if we substitute `a` for `b` in the following program, the resulting program is *not* bisimular to the original:

```
MODULE main
VAR
  p : process b;
ASSIGN
  next(p.x) := 1;
```

This is because process `main` may intervene between steps of process `p`, changing the value of `p.x` to 1. In this state, which is not reachable in `a` or `b` alone, the two modules have different behaviors. Hence parallel composition in SMV.1 does not respect bisimulation (neither does it respect simulation). This problem is a general feature of languages that support interleaving processes with shared variables. It is difficult, for example, to formulate a compositional rule for the *leads-to* operator of UNITY logic [CM88]. For this reason, we will use only the SMV.0 subset for induction over processes.

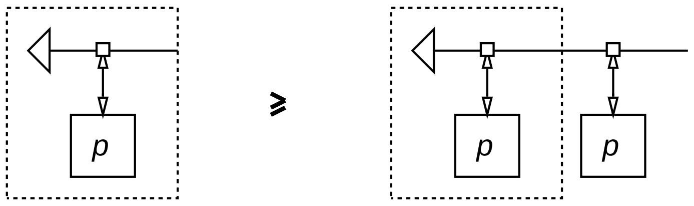


Figure 5.2: Substitution generating processors on bus

5.3 Example: The Gigamax protocol

In this section, we formulate a safe substitution rule that generates an arbitrary number of Gigamax processors attached to a cluster bus. As our invariant, we will use a single processor module attached to the end of a cluster bus. The strategy will be to generalize this module by adding nondeterminstic choice until it is able to simulate itself with one additional processor attached, as viewed from the bus. The counterexamples produced by the model checker will provide clues as to how the proposed invariant should be generalized. After a correct invariant is obtained, we can use this invariant to prove properties of the protocol that hold independent of the number of processors on a cluster bus.

The general form of the substitution rule we use is depicted in figure 5.2. The SMV code representing the left hand side is shown in figure figure 5.3, and the code for the right hand side (the invariant) is shown in figure 5.4. In our first guess for the invariant, we will use the original processor model from the previous chapter. Our approach will be to add behaviors (*ie.*, non-determinism) to the processor model until we have a correct invariant.

Essentially, we are testing whether one processor can mimic the actions of two processors as seen from the bus. Checking this produces a counterexample in which one of the two processors reaches the owned state, then the second processor issues a read command. This behavior cannot be produced by a single processor. To fix this problem, we can modify the processor model so that a processor is allowed to issue a read command in the owned state. It then sets its own "snoop" flag, and enters the shared state on a read-shared, and the owned state on a read-owned. Testing this new invariant produces another counterex-

```
MODULE rule(CMD,REPLY-OWNED,REPLY-WAITING,REPLY-STALL,
            cmd,reply-owned,reply-waiting,reply-stall,master)
VAR
  b : bus-connector(self,p);
  p : processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
  q : invariant(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL,
   b.cmd,b.reply-owned,b.reply-waiting,b.reply-stall,b.master);

MODULE bus-connector(a,b)
ASSIGN b.master := !a.master union 0;
DEFINE
  cmd :=
    case
      a.master : a.cmd;
      b.master : b.cmd;
      1 : idle;
    esac;
  reply-owned := a.reply-owned | b.reply-owned;
  reply-waiting := a.reply-waiting | b.reply-waiting1;
  reply-stall := a.reply-stall | b.reply-stall1;
  master := a.master | b.master;
```

Figure 5.3: Substituion rule for adding one processor

```
OPAQUE MODULE invariant(CMD,REPLY-OWNED,REPLY-WAITING,
  REPLY-STALL,cmd,reply-owned,reply-waiting,reply-stall,master)
SIMULATES rule
VAR
  b : bus-connector(self,p);
  p : processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
  t : bus-terminator(CMD,REPLY-OWNED,REPLY-WAITING,REPLY-STALL,
    b.cmd,b.reply-owned,b.reply-waiting,b.reply-stall,b.master);

MODULE bus-terminator(CMD,REPLY-OWNED,REPLY-WAITING,
  REPLY-STALL,cmd,reply-owned,reply-waiting,reply-stall,master);
ASSIGN
  CMD := cmd;
  REPLY-OWNED := reply-owned;
  REPLY-WAITING := reply-waiting;
  REPLY-STALL := reply-stall;
```

Figure 5.4: The invariant

ample in which the first processor reaches the owned state, then issues a read command (thus setting its snoop and waiting bits), then the second processor issues a read command. One processor alone cannot produce this behavior, since it cannot issue a second read command while its waiting flag is set. We modify the processor model to allow this behavior. Note that this is behavior is safe, since the second read command is blocked by the waiting flag which is already set. With this modification, we have a correct invariant.

Using this invariant, we can check properties of the system in $\forall$-CTL, using the invariant in place of the processors on the cluster buses. The substitution rule can be applied as many times as necessary to produce a system with an arbitrary number of processors while preserving all of the verified properties. We can also refer these properties back to our original model by showing that the generalized processor model simulates the original one. In order to verify properties such as deadlock freedom, however, which use existential path quantifiers, it would be necessary to prove bisimulation rather than simulation. This means that it would not be possible to use the strategy of generalizing the original model until an invariant is reached, since the generalized model would not be bisumulation equivalent to the original model.

5.4 Related research

A number of methods have been proposed in the past for extending automatic verification to parameterized designs that have an arbitrary number of similar or identical processes.

The first to approach this question were Browne, Clarke and Grumberg [BCG86], who extended the logic CTL to a logic called *indexed* CTL. This logic allows the restricted use of process quantifiers as in the formula $\bigvee_i f(i)$, which means that the formula f holds for some process i. Restricting the use of these quantifiers and eliminating the next-time operator makes it impossible to write a formula which can distinguish the number of processes in a system. By establishing an appropriate equivalence between a system with n processes and a system with $n+1$ processes, one can guarantee that all systems satisfy the same set of formulas in the indexed logic. This method was used to establish the correctness of a mutual exclusion algorithm by exhibiting a bisimulation relation between an n-process system and a 2-processes system, and applying model checking to the 2-process system.

A disadvantage of the indexed CTL method is that the bisimulation relation must be proved in an *ad hoc* manner. Finite state methods cannot be used to check it because it is a relation between a finite-state process and a process with an arbitrary number of states. Clarke and Grumberg dealt with the problem of establishing a bisimulation by introducing the notion of a *process closure* P^*. This process must be derived by hand, and have the property that $M_r \parallel P^*$ is equivalent to $M_{r+1} \parallel P^*$ for some small r. This can be verified mechanically. Shtadler and Grumberg took this notion a step further by introducing *network grammars* to describe classes of finite state systems. This technique used an indexed form of linear temporal logic, and required that the processes on the left and right hand sides of each grammar rule be equivalent in an appropriate sense.

The requirement that all systems generated by the grammar be equivalent seems to be a rather strict limitation, however. The method of this chapter, which uses a partial order rather than an equivalence, was first proposed by Kurshan and McMillan [KM89], and simultaneously by Wolper and Lovinfosse [WL89]. Around the same time, Burch was also applying a similar idea to Dill's trace theory for speed-independent circuits.[1]

Another method for proving properties of systems of identical processes is due to German and Sistla [GS]. It uses a linear-time temporal logic for specifications (again, the next-time operator is not allowed) and is fully automatic. By means of a distinguished "control" process, it is possible to check some global properties (although process quantifiers are not present in the logic). Unfortunately, because the decision algorithm is doubly exponential in the process size, this method has not been applied in practice.

A system called GORMEL has been created by Marelly and Grumberg, implementing the techniques of [SG89]. GORMEL uses context free grammars to describe systems of processes. This is fairly similar to the use of module substutution rules in SMV. There are a number of differences between the systems, however. GORMEL is oriented towards verification of distributed algorithms. It uses a model of transition systems with pairwise synchronized actions, as in CCS. This model is not well suited for describing digital systems – first because most signals in hardware are broadcast to more than one location, and second because many signals are exchanged back and forth between components of a system in a single clock cycle. The difficulty of reducing this two way exchange of many signals to a single atomic action would make it extremely cumbersome to create a CCS-like model for a system like the Gigamax.

Another difference is in the logic – GORMEL uses an indexed version of LTL without next-time called LTL^2. As in indexed CTL, it is not possible to nest process quantifiers. Because of the ability to use process quantifiers, it is possible to express some properties which are not expressible in CTL, for example that if a proposition p is true in some process, then it is eventually true in all processes.

For the process relation, GORMEL uses a form of stuttering equiv-

[1]Personal communication

alence rather than simulation. This places fairly strong requirements on the allowable grammar rules. In particular, it is not possible in such a system to take the approach taken here of successively generalizing a component process in order to obtain an invariant, since the required relation between the left and right hand sides of the grammar rule is a symmetric one. The GORMEL approach will work if the various systems generated by the grammar can be distinguished only by stuttering (arbitrary repetition of the same state labeling).

A final difference between the systems is, of course, that SMV is based on symbolic model checking methods. This is not clearly an advantage, however, since the state explosion problem may not be very severe for the small number of processes that tend to be involved in induction rules.

Chapter 6

A partial order approach

In this chapter, we consider an alternative to the symbolic model checking method which is also aimed at avoiding the state explosion problem. A number of researchers have observed that the arbitrary interleaving of concurrent actions is a major contributor to the state explosion problem, and that substantial efficiencies could be obtained if the enumeration of all possible interleavings could be avoided. As a result, several have proposed verification algorithms based on partial orders [Val89, Val90, God90, GW91, PL89, PL90, YTK91]. The method presented here is based on unfolding a Petri net into an acyclic structure called an *occurrence net*. The notion of unfolding was introduced by Nielsen, Plotkin and Winskel as a means for giving a concurrent semantics to nets, but in this case the goal is to avoid the state explosion problem. An algorithm is introduced for constructing the unfolding of a net, which terminates when the unfolded net represents all of the reachable states of the original net. The unfolding is therefore adequate for testing reachability (to be more precise, *coverability*) and deadlock properties. It is shown using an asynchronous circuit example that the unfolding can be polynomial in the circuit size while the state space is exponential. In contrast, the stubborn sets method of Valmari [Val89, Val90] and trace automaton method of Godefroid [God90, GW91] are ineffective in reducing the state explosion problem for asynchronous circuit models, because of the ubiquity of confusion in such models. In addition, because the unfolding method is fully automatic, it has a certain advantage over behavior machines method of Probst [PL89, PL90], which requires a pomset grammar describing the circuit's behavior to be constructed by hand.

6.1 The unfolding operation

Briefly, an occurrence net is a Petri net without backward conflict (two transitions outputting to the same place), and without cycles. Such a net can be obtained from an ordinary place/transition net by an unfolding process. Figure 6.1 shows an example of a net and part of its unfolding. Since the occurrence net it is acyclic and rooted, there is a natural well founded (partial) order on the transitions and places of the net. This order is called the dependency order. It is impossible for a transition of the occurrence net to fire unless all of its predecessors in the dependency order have fired.

The most important theoretical notion regarding occurrence nets is that of a *configuration*. A configuration represents a possible partial run of the net – it is any set of transitions that satisfies the following conditions:

1. If any transition is in the configuration, then so are all of its predecessors in the dependency order (a configuration is *downward closed*).

2. A configuration cannot contain two transitions in *conflict*, meaning that both input from the same place.

An example of a configuration is shown in figure 6.2, with elements of the configuration filled in black. Two transitions in the figure are hatched in. Either of these transitions can be added to the black set to form a new configuration. Adding any other transition would be illegal, however, since it would either violate downward closure or conflict-freeness.

In an unfolding, each transition corresponds to a transition of the original net, and each place corresponds to a place of the original net. We can associate each configuration of the unfolding with a state (marking) of the original net by simply identifying those places whose tokens are produced but not consumed by the transitions in the configuration.

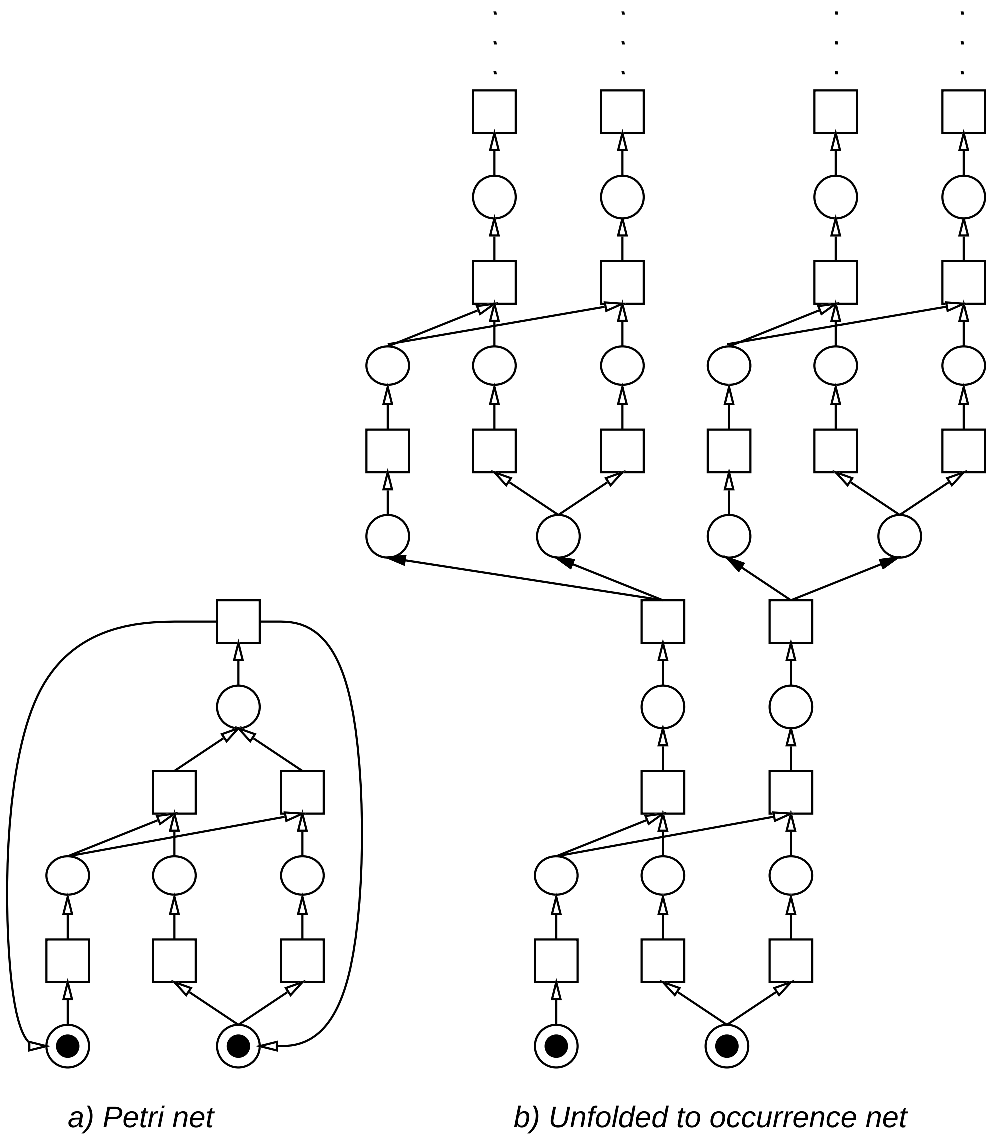


Figure 6.1: Unfolding example.

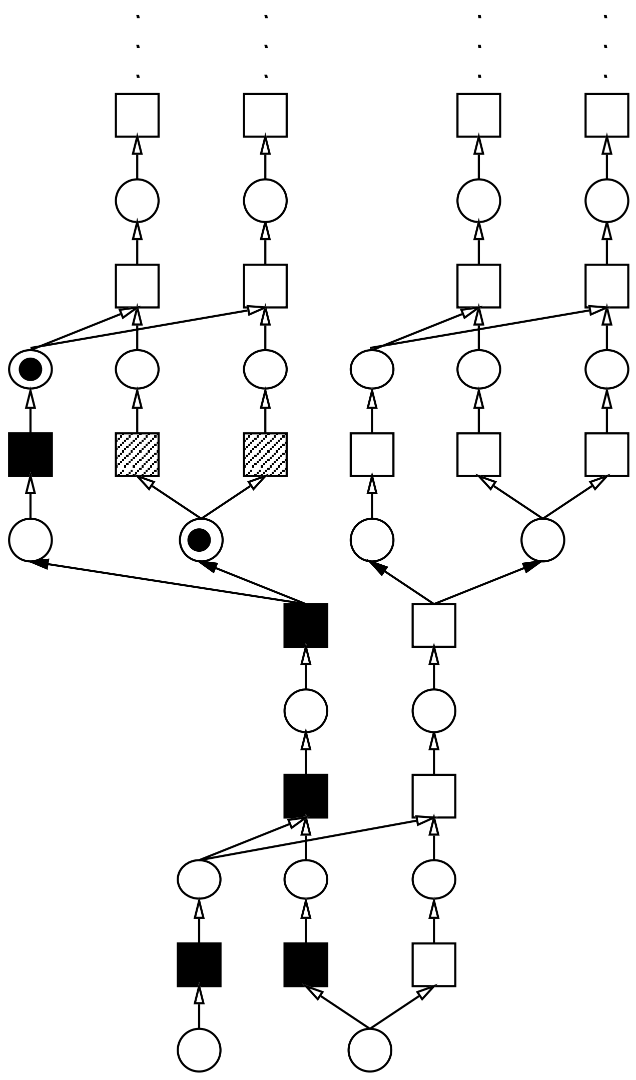


Figure 6.2: Configuration

This set is marked with black dots in figure 6.2. Mapping this set back onto the original net, we obtain the *final state* of the configuration.

The final theoretical notion we need regarding unfoldings is that of a *local configuration*. The local configuration associated with any transition consists of that transition and all of its predecessors in the dependency order (that is, the downward closure of the transition as a singleton). This is the set of transitions which necessarily are contained in any configuration containing the given transition. Note that a local configuration may not exist if this set contains two transitions in conflict.

We are now ready to consider the problem of building a fragment of the unfolding which is large enough to represent all of the reachable markings of the original net. Building the unfolding itself is straightforward. The process starts with a set of places corresponding to the initial marking of the original net. The unfolding is grown by finding a set of places in the unfolding which correspond to the inputs (preset) of a transition in the original net, then adding a new instance of that transition to the unfolding, as well as a new set of places corresponding to its outputs (postset). If the new transition has no conflicts in its local configuration (more precisely, if it *has* a local configuration) it is kept, otherwise it is discarded. This is because the existence of a conflict means that the new transition can occur in no configurations of the unfolding.

The key to termination of the unfolding is to identify a set of transitions of the unfolding to act as *cutoff points*. This set must have the following property: any configuration containing a cutoff point must be equivalent (in terms of final state) to some configuration containing no cutoff points. From this definition, it follows that any successor of a cutoff point can be safely omitted from the unfolding, without sacrificing any reachable markings of the original net. To see this, suppose we have built the unfolding only up to the cutoff points, in the sense that any new transition we can add must have a cutoff point as a predecessor. From this point on, any transition we add must be descended from some cutoff point. Thus, any configuration we might add to the unfolding must have the same final state as some configuration already present.

A sufficient condition for a transition to be a cutoff point is the following: the final state of its local configuration is the same as that of some other transition whose local configuration is smaller. The proof of this statement is as follows: suppose there are two transitions t_1 and t_2, whose local configurations have the same final state, with that of t_2 being smaller. Now imagine a configuration C_1 (local or otherwise) containing t_1. We can obtain C_1 from the local configuration of t_1 by adding the transitions in the difference one at a time, in an order consistent with the dependency relation. According to our construction, at each step of this process, there is a corresponding transition we can add to the local configuration of t_2 leading to the same final state. Hence, we can build a configuration C_2 containing t_2 which has the same final state, *but is at least one transition smaller than* C_1, since we started from a smaller set. Thus if any configuration contains a cutoff point, it is equivalent to a smaller configuration. Configurations cannot be made arbitrarily small, however, so any configuration containing a cutoff point must be equivalent to a configuration not containing a cutoff point. Since all the reachable states are represented by configurations containing no cutoff points, it is unnecessary to build the unfolding beyond any cutoff point.

We can find the cutoff points by simply keeping a hash table of all transitions, indexed by the final state of the local configuration. If when generating a transition, we find in the table a transition with equivalent but smaller local configuration, we discard the new transition. We can show, as follows, that this process is guaranteed to terminate if the original net is bounded and finite. First, the depth of the unfolding must be bounded by the number of number of reachable markings. The depth of a given transition in the unfolding is the longest chain of predecessors of that transition. Each transition in this chain has a local configuration, and these local configurations form a chain of increasing size. If the depth of the given transition is greater than the number of reachable markings of the original net, then by the pidgeon-hole principle, two of these local configurations must have the same final state. This cannot be, however, since in this case one of the transitions in the chain would have been determined to be a cutoff point. If the original net is bounded, it has a finite number of reachable markings, hence the depth of the unfolding is bounded. If the original net is finite, we can show by induction that the number of transitions at any given

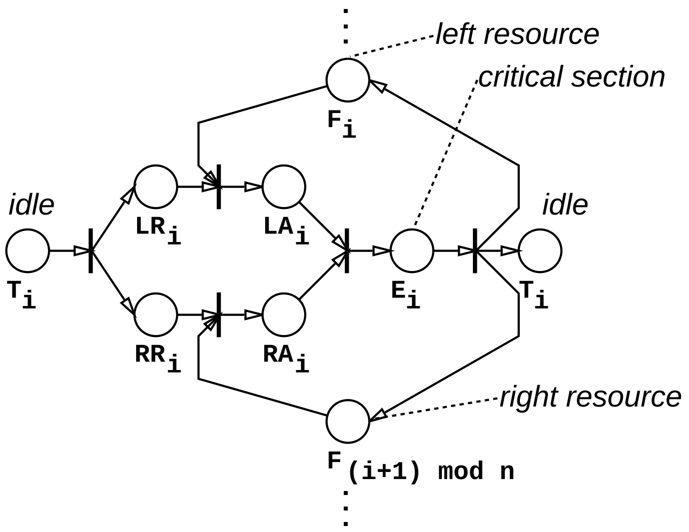


Figure 6.3: Dining philosophers net.

depth in the unfolding is finite. Hence the total number of transitions generated by the unfolding process is finite.

As an example of termination, consider the net of figure 6.3, which represents the dining philosophers paradigm. In this scenario, there are n concurrent processes (philosophers), each of which must acquire the use of two shared resources (forks) in order to execute its critical section (eating spaghetti). The processes are organized in a ring, with each neighboring pair sharing one resource. Figure 6.4 shows the completed unfolding for the case of three philosophers ($n = 3$). The cutoff points are marked with an X. The local configuration of each of these transitions is equivalent to the empty configuration. We observe that the size of the unfolding is not only bounded, but is linear in the number of philosophers, while the number of states is exponential as shown in table 6.1.

Recall that in growing the unfolding, it is necessary to enumerate all of the subsets of places which correspond to the inputs of transitions. The complexity of this is $O\binom{n}{i}$, where n is the size of the unfolding, and i is the largest number of inputs of any transition. This is,of course, bounded by n^i, which is polynomial given a fixed value of i. In practice,

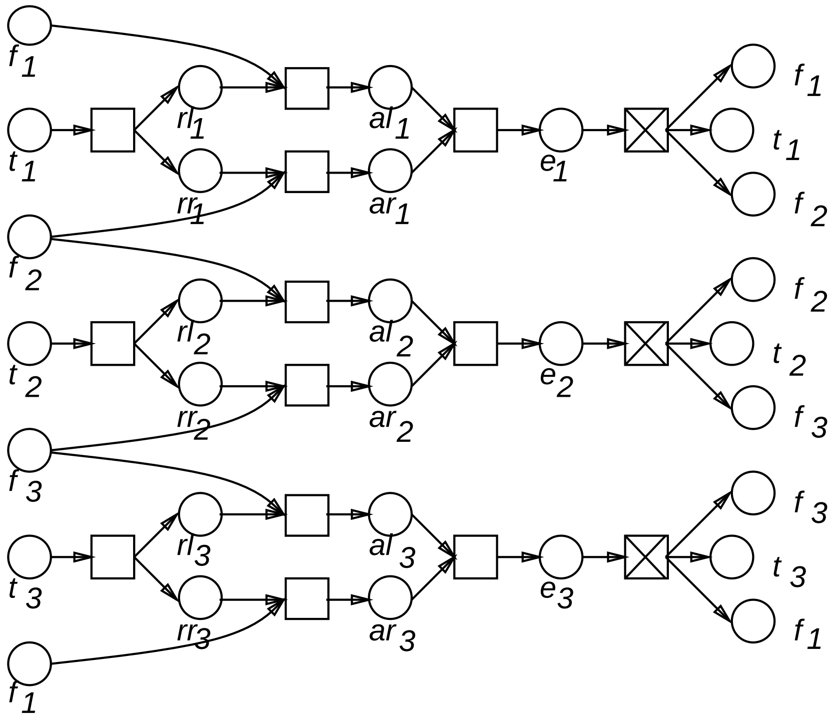


Figure 6.4: Unfolding of the dining philosophers net.

number of philosophers	size of unfolding (transitions)	number of reachable states
2	9	22
3	13	100
4	17	466
5	21	2164

Table 6.1: Unfolding size and number of states for Dining Philosophers

however, the number of subsets which are considered can be reduced quite effectively, using the following two techniques. First, suppose we are enumerating the subsets: we need not add any place to the set if the result would not be contained in the set of inputs of any transition. Second whenever a place is added to the set, we can immediately eliminate from consideration all of the places which have a predecessor in conflict with a predecessor of the new element, since any transition with both places as inputs would be discarded. We add transitions to the net in order increasing size of the local configuration, so that we can use a hash table to determine whether or not each transition is a cutoff point. Thus, whenever a candidate for a transition in the unfolding is generated, it is placed in a queue ordered by increasing local configuration size. The places of the net are enumerated by pulling the first element t' from this queue, testing whether it is a cutoff point, and if not, generating places for its outputs. The procedure terminates when the queue of candidate transitions becomes empty. Figures 6.5 and 6.6 show a pseudo-code implementation of this procedure. The pseudo-code is written somewhat inefficiently in places for simplicity.

In function Unfold, the arguments P, T and M_0 are the places, transitions and initial marking of the original net. Each place in the unfolding is represented by a pair $(place, preds)$, where *place* is the corresponding place in the original net, and *preds* is the set of immediate predecessor transitions in the unfolding (note that since there is no backward conflict, the size of this set is at most one). Each transition in the unfolding is represented by a pair $(trans, preds)$, where *trans* is the corresponding transition in the original net, and *preds* is the set of immediate predecessor places in the unfolding. The function returns P' and T', the set of places and transitions, respectively, of the unfolding. There is also a queue Q' of transitions to be expanded, and a hash table (HashTable) used for identifying cutoff points.

Coverability problems can be solved using the unfolding in the following way. Imagine we have a set of places in the original net, and we wish to determined whether this set can every be simultaneously marked. We simply add a new transition to the net, whose inputs are the given set, and then construct the unfolding. If the unfolding contains any instance of this new transition, the set is coverable, and otherwise not.

```
global $P'$,$T'$,$Q'$,HashTable[]
function Unfold($P$,$T$,$M_0$)
    $P' = T' = Q' = \emptyset$; clear HashTable
    for each $p \in M_0$ do
        add $p' = (p, \emptyset)$ to $P'$
        GenTrans($\{p'\}, T$)
    end for
    while the queue $Q'$ is not empty do
        pull the first $t'$ off of $Q'$
        if not IsCutoffPoint?($t'$) do
            for each $p$ in outputs of $trans(t')$ do
                add $p' = (p, \{t'\})$ to $P'$
                GenTrans($\{p'\}, T$)
            end for
        end if
    end while
    return($P'$,$T'$)
end function

procedure GenTrans($S', T$)
    if not exists $t \in T$ such that $place(S') \subseteq$ inputs of $t$ then return
    if Predecessors($S'$) has forward conflict then return
    forall $t \in T$ do if $place(S') =$ inputs of $t$ then
        add $t' = (t, S')$ to set $T'$
        insert $t'$ in $Q'$ in order of |LocalConfig($t'$)|
    end for
    for all $p' \in P$ where $p'$ older than any member of $S'$ do
        GenTrans($S' \cup p', T$)
end procedure
```

Figure 6.5: Pseudo-code implementation of unfolding procedure

```
function IsCutoffPoint?($t'_1$)
    $C'_1$ = LocalConfig($t'_1$)
    $S'_1$ = FinalState($C'_1$)
    $L'$ = HashTable[HashFun($S'_1$)]
    forall $t'_2$ in $L'$ do
        $C'_2$ = LocalConfig($t'_2$)
        if $S'_1$ = FinalState($C'_2$) and Size($C'_2$) $<$ Size($C'_1$) then return(1)
    end for
    add $t'_1$ to HashTable[HashFun($S'_1$)]
    return(0)
end function

function LocalConfig($t'$)
    return(Predecessors($\{t'\}$) $\cap T'$)
end

function Predecessors($S'$)
    do
        $S' = S' \cup$ preds($S'$)
    until $S'$ unchanged
end function

function FinalState($C'$)
    let $S'$ be the set of all $p' \in P'$ such that preds($p'$) $\subseteq C'$
    return(place($S'$ $-$ preds($C'$)))
end function
```

Figure 6.6: Pseudo-code, continued.

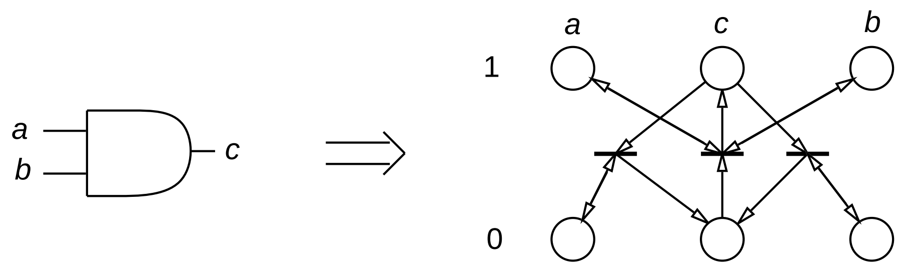


Figure 6.7: Translation from circuit to net

6.2 Application example

We now consider a more realistic example than the dining philosophers – a speed-independent [Sei80b] circuit designed to implement a distributed mutual exclusion (DME) protocol. The circuit was designed by Alain Martin [Mar85] and has been analyzed using an abstracted trace theoretic model by Dill [Dil88]. It was also used as an example for the symbolic model checking method in section 2.4.2.

Networks of logic gates in speed-independent circuits are readily modeled by Petri nets. A network of n gates can be modeled by a Petri net of $O(n)$ places. When we model a network of gates as a Petri net, we introduce two places for each input of each gate. One represents the the input in a logic low state, while the other represents the input in a logic high state. Transitions in the Petri net correspond to rising or falling transitions of gate outputs. A rising transition of a gate output removes all the logic low tokens from the inputs to which it is connected, and places tokens on the corresponding logic high places.

As an example, figure 6.7 shows the net fragment representing an AND gate. When both inputs of the gate are at the logic high state, we can move a token from the place representing logic low at the output to the place representing logic high. Similarly, if either input is at the logic low state, we can move a token from the place representing logic high at the output to the place representing logic low.

A dynamic hazard occurs, for example, if the AND gate's output is enabled to rise while one of the inputs is enabled to fall. The problem of whether or not a dynamic hazard can occur can thus be posed as a coverability problem. Alternatively, since dynamic hazards correspond to dynamic conflicts in the unfolding, the problem can be solved by constructing the unfolding and examining it for dynamic conflicts, *ie.*, two transitions which are in conflict, and which may be simultaneously enabled. The DME circuit also uses special two-way mutual exclusion elements as components, which are immune to certain hazards. In checking the DME ring for hazards, we ignore conflicts between rising transitions of a mutual exclusion element's acknowledge outputs.

Table 6.2 shows the results of the occurrence net unfolding procedure (ONU), and a depth-first traversal of the state space (DFT), for the Petri net model of the circuit, for rings with one to five cells. The depth of the occurrence net unfolding for the case of 5 cells was 141 transitions. The number of transitions in the ONU increases quadratically in the number of cells. This is because as the number of cells in the ring increases, a request must be relayed through a greater number of stages in order to obtain the token, in the worst case. At the same time, the number of cells which are requesting also increases. The occurrence net therefore grows in both width and depth in proportion to the number of cells. As we increase the number of cells in the ring, the number of reachable global markings increases exponentially. For this reason, it was only possible to apply DFT to a system of five cells, before the available memory resources were exhausted. It is known, however, from using OBDD based methods, that the number of states increases asymptotically by slightly less than a factor ten for each added cell.

How do these results compare other methods for avoiding the state explosion problem? The trace theory approach of Dill [Dil88] required an abstract model of the arbiter cell to be created by hand. This reduces the state explosion problem, but does not entirely solve it, since even with the reduced model, the number of states still increases exponentially with the number of components. Probst [PL90] reports a method which requires quadratic space and time in the number of cells, but also is not fully automatic. The methods of Valmari [Val89, Val90] and Godefroid [God90, GW91] and Yoneda [YTK91] cannot be effectively applied to this example or to other speed independent circuits, because in all states, all enabled transitions are in conflict with some disabled transition. Thus no transition can be statically guaranteed to be per-

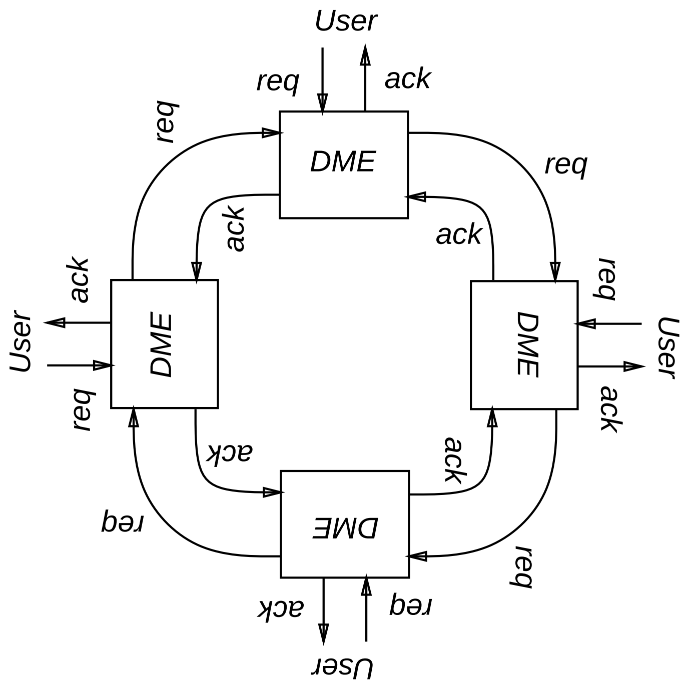


Figure 6.8: Distributed mutual exclusion circuit

sistent. Experiments by Holger Schlingloff[1] have confirmed this to be the case. It is possible, perhaps, that some more clever static analysis technique could be used to show that some transitions are persistent, in which case these methods could be applied to some effect.

Finally, in chapter 2, we saw that the symbolic model checking method had cubic time complexity and linear space complexity, using a simultaneous model. Burch and Long[2] have obtained $O(n^{2.5})$ time complexity for this circuit using symbolic model checking with a modified search order (cf. section 2.8). This method requires some hand optimization, however. In any event, it appears that the symbolic model checking method yields somewhat better asymptotic performance for the DME circuit, though both methods effectively solve the state explosion problem.

[1]Personal communication

[2]Personal communication

number of cells	size of unfolding (transitions)	number of reachable states
1	23	22
2	125	502
3	313	6579
4	604	75172
5	1013	802425

Table 6.2: Performance of ONU and DFT on hazard-detection problem for the distributed mutual exclusion circuit

6.3 Deadlock and occurrence nets

Besides coverability, another interesting problem for Petri nets is the question of deadlock. A *terminal marking* of a Petri net is one in which no transitions are enabled. Reachability of a terminal (or deadlocked) state cannot be framed in terms of the coverability problem. However, since the unfolding represents all reachable markings, a net has a reachable terminal marking if and only if its unfolding has a reachable terminal marking. The problem of existence of a reachable terminal marking of an occurrence net is NP-complete. This is easily shown by reduction from 3-SAT.[3] To see this consider the formula $(x_1 + y_1 + z_1)(x_2 + y_2 + z_2) \cdots (x_n + y_n + z_n)$ where each x_i, y_i and z_i is a positive or negative literal. Assume the formula has m variables. Let the positive literals be $l_1, \ldots, l_m$, and the negative literals be $\bar{l}_1, \ldots, \bar{l}_m$. In polynomial time, we can construct a net which has a terminal marking if and only if the formula is satisfiable. The initial marking of the net is a set of places $\{v_1, \ldots, v_m\}$. There is a place representing each positive literal $l_1, \ldots, l_m$ and each negative literal $\bar{l}_1, \ldots, \bar{l}_m$. For each variable v_i, there is a transition from v_i to l_i and from v_i to $\bar{l}_i$. For each conjunct $(x_i + y_i + z_i)$, there is a transition c_i, whose preset is $\{\bar{x}_i, \bar{y}_i, \bar{z}_i\}$. In other words, the transition c_i is enabled to fire if and only if $(x_i + y_i + z_i)$ is false. Thus, some transition c_i is enabled to fire if and only if the whole formula is false. The postset of each transition

[3]Satisfiability of a Boolean formula in conjunctive normal form, with three literals in each conjunct.

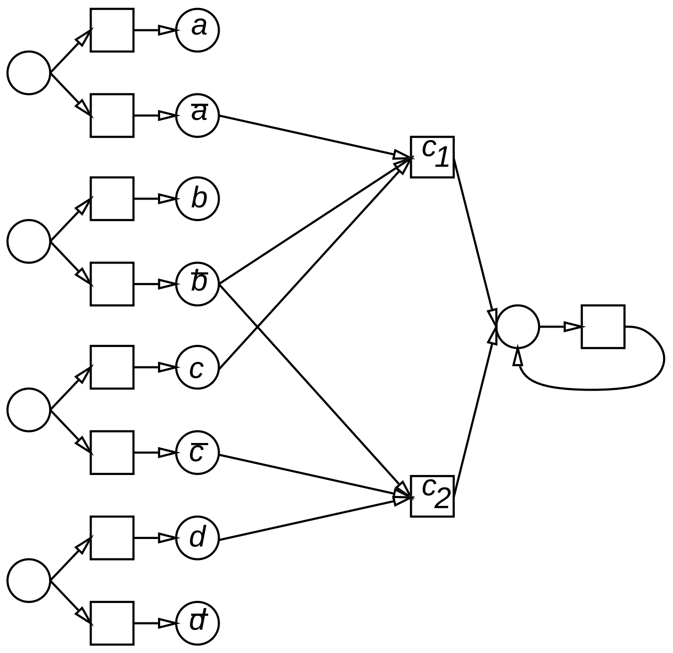


Figure 6.9: Reduction from 3-SAT problem to a terminal marking problem.

c_i is the single place $\{q\}$, and there is a transition from $\{q\}$ to $\{q\}$. Thus, if any c_i fires, the net may never reach a terminal marking. As a result, there is a terminal marking of the net if and only if the formula is satisfiable. For example, figure 6.9 shows the net constructed for the formula $(a + b + \bar{c})(b + c + \bar{d})$.

The reader may easily verify that the size of the unfolding of such a net (up to the cutoff points) is linear in the size of the original net. In fact, it is essentially the same net, except the the place q occurs n times in the unfolding. Since all reachable markings of the original net occur as configurations of the unfolding, the unfolding has a terminal marking if and only if the formula is satisfiable. Hence 3-SAT is P-time reducible to reachability of a terminal marking of an unfolding. Since the configuration representing the terminal marking can be guessed in P-time in the size of the unfolding, and also tested in P-time, it follows that the problem is in NP, and hence NP-complete.

Interestingly, however, the problem is readily solved in practice even

```
let $B$ be the set of the cutoff points, $T_s = \emptyset$
while $B$ is not empty do
  let $t$ the the element of $B$ with the fewest spoilers
  if $t$ has no spoilers, then backtrack
  choose an element $t'$ from the spoilers of $t$
  add $t'$ to $T_s$
  delete all transitions in conflict with $T_s$
end do
```

Figure 6.10: Procedure to detect terminal marking.

for very large unfoldings, using an algorithm based on techniques of constraint satisfaction search. The key observation which leads to this algorithm is that there is no terminal marking exactly when all configurations the unfolding can reach some configuration containing a cutoff point. This is simply because if there is no terminal marking, then all configurations can reach a configuration which is arbitrarily large. A configuration C' can reach a configuration containing transition t' if and only if the union of C' and the local configuration of t' is a configuration. If it is not, then no set containing C' and t' is a configuration. If the union is not a configuration, we will say that C' and t' are in conflict. Hence, there is a terminal marking if and only if there is a configuration which is in conflict with every cutoff point. The search for such a configuration can be carried out using branch and bound techniques. For example, if a configuration C' is in conflict with a cutoff point t', there must be a transition $t'_1 \in C'$ which is in conflict with some transition in the local configuration of t'. Such a transition t'_1 will be called a *spoiler* of t'.

There exists a configuration in conflict with all of the all of the cutoff points (equivalently, there exists a terminal marking) if and only if there exists a configuration containing a spoiler for every cutoff point. The set of spoilers contained in this configuration will be called T_s. The algorithm of figure 6.10 uses branch and bound techniques to find such a set T_s if one exists.

Note that in line 3 of the procedure, the cutoff point with the smallest number of spoilers is chosen so that the number of choices in line 5 is minimized. Whenever a spoiler for a given cutoff point is chosen to belong to T_s in line 5, everything in conflict with T_s is eliminated from future consideration in line 7. Note that the cutoff points in conflict with T_s are also eliminated, which cuts down on the amount of future branching. Whenever there is a cutoff point with no remaining spoilers, the procedure backtracks, from line 4 to the most recent occurrence of line 5 where there are remaining choices. If there are no remaining choices, the procedure fails. Of course, when backtracking occurs, the the net is also returned to the state it was in at the point where execution is being resumed. This backtracking is easily implemented by keeping a stack of the remaining choices for t' in each iteration of the loop, and marking each transition in the net with the level of the stack at the time it was "removed". Interestingly, if the procedure terminates successfully, the remaining net has the property that every path leads to a terminal marking of the original net N. This makes it straightforward to extract a path leading to a terminal marking.

Obviously, because of the backtracking, this procedure is exponential (as it must be, if $\mathcal{P} \neq \mathcal{NP}$). However, this is only the worst case. The dining philosophers serve as an example of a case in which the exponential complexity is avoided. In fact, the procedure finds the terminal marking in time which is *linear* in the number of philosophers. This is easily seen by examining the unfolding of the Dining Philosophers net in figure 6.4. There is one cutoff point in this net for each process. Initially, each of these transitions has two spoilers, which correspond to the two resources required to enter the critical region being granted to the two neighboring processes. Regardless of which cutoff point is used first, the symmetry is then broken as the part of the net in conflict with one of the two spoilers is removed. This removes, in particular, the transition which granted one of the resources to the first philosopher, hence one of its neighbors now has only one spoiler, so there is only one choice available the next time line 5 is reached. After this spoiler is added to T_s, the remaining neighbor of the second philosopher now has only one spoiler. This process continues without backtracking until it has come full circle and the terminal marking is found. Note that if the cutoff point with the fewest spoilers were not chosen in line 3, the procedure might have examined an exponential number of candidates for T_s before a valid one was found.

In fact, using nets representing communication protocols as examples, this procedure has been successfully been applied to unfoldings with more than 3000 transitions and 1000 cutoff points, where some cutoff points had as many as 50 spoilers. It is clear that the branch and bound technique quickly narrows down the number of choices in these examples.

6.4 Relation to AI techniques

The occurrence net unfolding method, as applied to the coverability problem, was inspired by so-called "least commitment" strategies for AI planning problems, especially nonlinear planning techniques. Like these strategies, the method falls into the category of searching in the space of solutions (*ie.*, covering sequences for a given set), rather than the space of the problem (*ie.*, the reachable markings). Each partially constructed unfolding represents some set of possible partial solutions which may be extended to a complete solution. As in other constraint satisfaction search methods, the method tries to eliminate as early as possible those partial solutions which cannot be extended to complete solutions. This is done in the unfolding procedure by the elimination of candidate transitions which have no local configuration, and also by the cutoff points, which effectively discard those partial solutions which cannot be extended to a lowest cost (*ie.*, fewest transition) solution. This is done without unnecessarily committing to the order of independent transitions. This makes the unfolding method somewhat similar to constraint posting methods used in non-linear planning. Both methods construct fairly similar structures, although non-linear planners, such as NOAH [Sac77] only represent one partial solution, while the occurrence net represents all partial solutions. Non-linear planners attempt to detect conflicts and eliminate them by posting additional constraints on the solution or by modifying elements of the solution. Non-linear planners also use heuristics to guide them towards a solution, and hence sometimes overconstrain the solution space and require backtracking. For this reason, they are heuristically efficient, but would not be suitable for exhausting the solution space, as is required in automatic verification methods. In fact, NOAH is not even guaranteed to find a solution where one exists. A later system called TWEAK [Cha87] does guarantee a solution where one exists, but fails to terminate if no solution exists. Since TWEAK does overconstrain the solution space, exhaustive search could only be achieved by backtracking, even if a suitable termination condition were found. The unfolding procedure never overconstrains the solution space, however, as conflicting transitions may coexist in the unfolding. Hence, it does not backtrack.

6.5 Evaluation

When is unfolding a suitable strategy for problems in automatic verification? The most promising application is hazard checking for asynchronous control circuits. In these circuits, the state explosion seems to derive almost entirely from arbitrary interleavings of concurrent transitions. In such cases, the unfolding method can have a considerable advantage over methods that search the entire state space. Note, however, that other methods based on partial orders are not necessarily effective in reducing the state explosion for these circuits, because of the aforementioned problem of determining when transitions of the net are persistent.

In general, any problem which can be posed in terms of coverability or deadlock in a Petri net model is a possible application of the unfolding method. In addition, it is possible that heuristically efficient procedures can be found for deciding the existence of an infinite firing path in some ω-regular set, given an unfolding. In this case, specifications framed as linear time temporal logic formulas, or ω-automata could be evaluated.

Chapter 7

Conclusion

What we have seen in the preceding chapters is that Ordered Binary Decision Diagrams can be used as a representation in a wide variety of automatic verification algorithms, in order to cope with the state explosion problem. This can be done in a unified way by representing the algorithms in the Mu-Calculus fixed point notation. For fairly diverse families of regularly structured systems, the CTL model checking algorithm was observed to run in time and space which increased polynomially in the size of the system, while the number of reachable states increased exponentially. These results bear out a theoretical result bounding the OBDD representation of the transition relation for such systems. Standard automatic verification algorithms would be unsuitable for these examples because their complexity is proportional to the number of reachable states.

Using OBDD based techniques, and a language suitable for the abstract modeling of digital systems, it was possible to verify a fairly complex industrial design for a cache consistency protocol, finding a number of subtle errors in the process. The verification process is valuable not only because of the advantages of formalization and exhaustive checking, but because it can find protocol errors more quickly than simulation, despite the exponential growth in states as the model increases in size. The ability to isolate high level errors quickly shortens the loop between design and verification, making it possible to experiment more freely with alternative designs, and shortening the “critical path” from conceptualization to implementation.

By a technique of induction over processes, it is possible to prove properties of a protocol which are independent of the number of processes participating in the protocol. This type of proof requires a sufficient understanding of the protocol on the part of the designer to construct a process invariant. Invariants are difficult to find, but the symbolic model checker provides an aid in this process by producing counterexamples for unsound invariants. In the author's opinion, finding a process invariant for a protocol is not only of value as a proof technique – the understanding of the protocol required to formulate the invariant can lead to simpler and more elegant protocols. This is another reason for formalizing and verifying a protocol before attempting to implement it.

The verification technique based on occurrence nets shows that OBDDs are not the only representation that can be used to avoid the state explosion problem. There are, in fact, certain advantages to the occurrence net based method for the example presented, since the memory usage is small, and no heuristic technique is required to produce a variable ordering. Still, at this stage, the occurrence net method is certainly not as well advanced as the symbolic model checking method.

There are several areas where the current work falls short of the goal of complete automatic verification of digital systems. In the case of the Gigamax protocol, an abstract model of the protocol was verified and not the actual implementation. Verification of the implementation would have been impossible due to a lack of formal models of the components of the system (*ie.*, standard devices, such as memories, registers, programmable logic, central processing units, *etc.*). If such models were available from the manufacturers, in principle the methods described in chapter 5 could be used to show that the implementation is simulated by the abstract model. Hierarchical reasoning of this kind has been extensively studied by Kurshan [Kur87]. Unfortunately, simulation does not preserve existential CTL properties such as absence of deadlock. As mentioned previously, bisimulation equivalence, which preserves all CTL properties, is too strong for this purpose, since the abstract models are necessarily non-deterministic, and the actual implementation cannot (and should not) exhibit this non-determinism. A practical technique of abstraction which preserves existential CTL properties is needed if existential properties are to be proved using hierarchical reasoning.

There is also a need for heuristic strategies for generating process invariants in inductive proofs. Marelly and Grumberg view the design of the invariant as part of the design of the protocol. This is a useful point of view, but some automated help beyond the generation of counterexamples would be useful for this purpose.

Finally, this work concentrates on how to solve the verification problem, once it has been formalized as the satisfaction of a temporal logic formula by a finite model, or as an appropriate relation between finite automata. There is, of course, a wide range of issues involved in formalizing the problem in the first place. For example, there is the ever present danger that the specification itself is incorrect. In the case of the very simple CTL formulas used to specify the Gigamax protocol, this is perhaps not a severe problem. The abstraction that was used to create a model for checking the sequential consistency property was, however, not obviously correct.

In general, there is a clear need for complete mechanical checking that the implementation of a processor or protocol matches the intended architecture (user model). This requires first of all a definitive model of the architecture – something that is currently lacking even for standardized architectures in the public domain. Second there must be a well defined criterion for determining what is a valid implementation of the architecture. Loosely, an implementation of a processor is equivalent to an architecture model if for all programs, the two machines produce the same "answer". However, for many reasons, this equivalence cannot be directly stated in terms of equivalence of finite state machines. For one, most modern CPU architectures have no explicitly defined notion of input and output. It is not adequate to view input and output as the sequence of loads or stores observed at the memory interface, since this sequence will differ among implementations (especially if the implementations contain cache memories, which is often the case). Solutions to the formalization problem are needed, but cannot be obtained by studying theoretical models alone. It is necessary to carefully consider what verification means in an engineering sense, as well as a mathematical sense.

Despite the shortcomings of current verification technology, it is clear that there are at least small areas of the problem space for which reasonable solutions exist, and these solutions can be put into practice to positive effect in an industrial setting. Those involved in verification research should perhaps take a closer look at engineering practice to determine how well the verification solutions match up with real engineering problems. This effort may lead not only to a more practical theory of formal verification, but also to a rich source of theoretical problems.

Appendix A

Semantics of SMV.1

This appendix defines the semantics of programs in the language SMV.1, which includes the subset SMV.0 plus the `process` keyword. In SMV.1, we need to account for both the arbitrary interleaving of processes, and the rules regarding when a variable may change value as a result of executing a given process.

A.1 The model

The set N of *names*, is the set of all character strings made up of the letters, the digits, the underscore and the minus sign characters, beginning with a letter. The *store* $L = L_V \cup L_H$ is made up of two disjoint, countably infinite sets of *locations* L_V and L_H. We will call the former the *visible* locations, and the latter the *hidden* locations. The set of locations L is defined recursively. It is the least set such that

1. if $n \in N$, then $n \in L_V$, and
2. if $l \in L_V$ and $n \in N$, then $l.n \in L_V$, and
3. if $l \in L_V$, then $.l \in L_H$.

The set of values V is the union of the integers in the range $[-2^{31}, 2^{31}-1]$ and N, the set of names. A *state* $x : L \to V$ is a function from locations to values. Let $S = L \to V$ be the set of all possible states.

If p is a declaration, then its denotation $[\![p]\!]$ is a quadruple (T, I, R, C). The T component is a partial function from L to the finite subsets of V. If l is a location, then $T(l)$, when defined, is the *type* of l – the set of values that can be assigned to location l. The component $I \subseteq S$ is the set of initial states. The component $R \subseteq S \times S$ is the transition relation. An asynchronous process is identified with a location r, which has the value 1 in a given state exactly when the process is executing in that state. The component $C \subseteq L \times L$ is the set of pairs (r, l) such that process r assigns the next value of location l.

A.2 Expressions

An expression denotes a function from states to finite subsets of V, according to the following rules:

1. If v is a value, then $[\![v]\!](x) = \{v\}$.
2. If l is a location, then $[\![l]\!](x) = \{x(l)\}$.
3. If e_1, e_2 are expressions, and o is one of

   `+, -, *, /, mod, >, >=, <, <=, =, &, |, ->, <->`

   then

   $$[\![e_1 \; o \; e_2]\!](x) = \{[\![o]\!](v_1, v_2) \mid v_1 \in [\![e_1]\!](x),\; v_2 \in [\![e_2]\!](x)\}$$

4. If e is an expression, then

   $$[\![!e]\!](x) = \{[\![!]\!](v) \mid v \in [\![e]\!](x)\}$$

5. If e_1, e_2 are expressions,

   $$[\![e_1 \; \texttt{union} \; e_2]\!](x) = [\![e_1]\!] \cup [\![e_2]\!]$$

6. If e_1, e_2 are expressions,

   $$[\![e_1 \; \texttt{in} \; e_2]\!](x) = [\![e_1]\!] \subseteq [\![e_2]\!]$$

The functions denoted by `+`, `-`, `*`, `/` are the usual functions of arithmetic modulo 2^{32}. The function denoted by `mod` is the positive remainder of division mod 2^{32}. The function denoted by the relational operators `>`, `>=`, `<` and `<=` return 0 when the relation is false and 1 when the relation is true, and are defined for numeric values only. For non-numeric values, they return $\perp$. The equality operator `=` is defined for all values, and returns 0 when they are unequal, and 1 when they are equal. The functions denoted by the Boolean operators are `&` (for and), `|` (for or), `!` (for not), `->` (for implies) and `<->` (for logical equivalence) are defined only for the values 0 and 1, and return $\perp$ otherwise.

A.3 Assignments and definitions

There is no semantic difference between assignments and definitions. If l is a location, and e is an expression, then the assignment $l := e$; denotes a quadruple (T, I, R, C), where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid l(x) \in e(x)\}$
4. $C = \emptyset$

The assignment `next`$(l) := e$; denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = S$
3. $R = \{(x, y) \in S^2 \mid x(\texttt{running}) \Rightarrow l(y) \in e(x)\}$
4. $C = \{(\texttt{running}, l)\}$

The assignment `init`$(l) := e$; denotes a triple (T, I, R) where

1. $T = \emptyset$
2. $I = \{x \in S \mid l(x) \in e(x)\}$
3. $R = S^2$
4. $C = \emptyset$

A.4 Variable declarations

If l is an identifier and $v_1, v_2, \ldots, v_n$ are values, then

$$\texttt{VAR}\ l\ :\ \{v_1, v_2, \ldots, v_n\};$$

denotes a quadruple (T, I, R, C) where

1. $T = \{(l, \{v_1, v_2, \ldots, v_n\})\}$
2. $I = \{x \in S \mid x(l) \in \{v_1, v_2, \ldots, v_n\})\}$
3. $R = \{(x, y) \in S \mid x(l), y(l) \in \{v_1, v_2, \ldots, v_n\})\}$
4. $C = \emptyset$

A.5 Renaming

Let $\phi : L \to L$ be a function from locations to locations. This in turn induces a map Φ on states, such that for all states x and locations l,

$$\Phi(x)(l) = x(\phi(l)).$$

If $M = (T, I, R, C)$, then let $\phi(M) = (T', I', R', C')$ where

1. $T'(\phi(l)) = T(l)$,
2. $I' = \{x \mid \Phi(x) \in I\}$ and
3. $R' = \{(x, y) \mid (\Phi(x), \Phi(y)) \in R\}$.
4. $C' = \{(\phi(r), \phi(l)) \mid (r, l) \in C\}$.

Note that the definition of T does not make sense if ϕ maps two locations with different types onto the same location. In this case, $\phi(M)$ is a type error.

A.6 Parallel composition

Let $M_1 = (T_1, I_1, R_1, C_1)$ and $M_2 = (T_2, I_2, R_2, C_2)$. Let n_1 and n_2 be two distinct names. For $i \in 1, 2$, let $\phi_i(l) = .n_i.l$ for all $l \in L_H$ and $\phi_i(l) = l$ otherwise, and let $M'_i = \phi(M_i)$. The parallel composition $M = M_1 \parallel M_2$ is defined as follows:

1. $T = T'_1 \cup T'_2$
2. $I = I'_1 \cap I'_2$
3. $R = R'_1 \cap R'_2$
4. $C = C'_1 \cup C'_2$

If $d_1, d_2, \ldots, d_k$ are declarations, then $[\![d_1 \; d_2 \; \ldots \; d_k]\!]$ is the parallel composition

$$[\![d_1]\!] \parallel [\![d_2]\!] \parallel \cdots \parallel [\![d_k]\!]$$

.

A.7 Instantiation

Suppose that module A is defined as follows:

$$\texttt{MODULE}\ A(n_1, n_2, \ldots, n_k)\ D$$

where $n_1, n_2, \ldots, n_k$ are distinct names and D is a sequence of declarations. We first consider the variable declaration $\texttt{VAR}\, r \; : \; A(l_1, l_2, \ldots, l_k);$ where $r, l_1, l_2, \ldots, l_k$ are visible locations. Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \leq i \leq k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. $\phi(.l) = .l$
3. for all $n \in N - \{\texttt{running}, n_1, n_2, \ldots, n_k\}$, $\phi(n) = r.n$, and $\phi(n.l) = r.n.l$,
4. $\phi(\texttt{running}) = \texttt{running}$, $\phi(\texttt{running}.l) = \texttt{running}.l$

Then $[\![\texttt{VAR}\ r\ :\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

Now we consider the declaration

$$\texttt{VAR}\ r\ :\ \texttt{process}\ A(l_1, l_2, \ldots, l_k);$$

Let ϕ be a renaming, such that, for all $l \in L_V$,

1. for all $1 \le i \le k$: $\phi(n_i) = l_i$, and $\phi(n_i.l) = l_i.l$,
2. for all $n \in N - \{n_1, n_2, \ldots, n_k\}$, $\phi(n) = r.n$, and $\phi(n.l) = .m_1.n.l$,
3. $\phi(.l) = .l$

Then $[\![\texttt{VAR}\ r\ :\ \texttt{process}\ A(l_1, l_2, \ldots, l_k);]\!] = \phi(D)$.

A.8 Programs and interleaving

Suppose that module `main` is defined as follows:

$$\texttt{MODULE main}\ D$$

where D is a sequence of declarations and $[\![D]\!] = (T, I, R, C)$. The number of processes executing in state x is

$$n_e(x) = |\{r | (x(r) = 1) \wedge \exists l :\ (r, l) \in C\}|.$$

The set of legal interleaving states is

$$S_I = \{x \in S \mid n_e(x) = 1\}.$$

The set of states in which location l is constrained to remain unchanged is

$$U(l) = \{x \in S \mid [\exists r : (r, l) \in C] \wedge \neg\exists r : [(r, l) \in C \wedge (x(r) = 1)]\}$$

The denotation of the program is a triple (T, I', R'), where

1. $I' = I \cap S_I$,
2. $R' = \{(x, y) \in R \mid x \in S_I \wedge \forall l \in L : (x \in U(l) \Rightarrow x(l) = y(l))\}$.

A.9 Specifications

Each program is associated with a Kripke structure which determines the truth value of CTL formulas in the specification. The atomic propositions in this case are all the Boolean valued expressions. The Kripke structure associated with a program whose denotation is the quadruple (T, I, R) is a Kripke model $K = (S, R, L')$ where

1. S is the set of states defined above,

2. R is the transition relation, and

3. if e is an expression, then

$$L'(e) = \{x \in S \mid [\![e]\!](x) = \{1\}\}$$

The specification is a formula f in CTL with fairness constraints. It is satisfied exactly when $K, s_0 \models f$ for all $s_0 \in I$.

Bibliography

[AB86] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, C-27(6):509–516, August 1978.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *ACM Symp. Principles of Programming Languages*, pages 164–176, 1981.

[BBB$^+$87] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *24th Design Automation Conference*, 1987.

[BBS90] Derek L. Beatty, Randal E. Bryant, and Carl-Johan H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, April 1990.

[BCD86] M. C. Browne, E. M. Clarke, and Dill. Automatic verification using temporal logic: Two new examples. In George J. Milne and P. A. Subramanyam, editors, *Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 113–124. North-Holland, 1986.

[BCDM86] 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, 1986.

[BCG86] M.C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. In *ACM Symp. Principles of Distributed Computing 5*, 1986.

[BCG87] M.C. Browne, E. M. Clarke, and O. Grumberg. Characterizing kripke structures in temporal logic. Technical Report 87-104, Carnegie-Mellon University. Computer Science Department, 1987.

[BCL91a] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, San Francisco, California, June 1991.

[BCL91b] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the IFIP International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[BF89a] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, 1989.

[BF89b] Soumitra Bose and Allan L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In Luc Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, pages 759–764, November 1989.

[Bil87] J. P. Billon. Perfect normal forms for discrete functions. Technical Report 87019, BULL, March 1987.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Bry88] Randal E. Bryant. Verifying a static ram design by logic simulation. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 335–349. MIT Press, 1988.

[Bry91] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Computers*, 40(2):205 – 213, February 1991.

[BS90] Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using symbolic ternary system models. In Robert Kurshan and Edmund M. Clarke, editors, *Workshop on Computer-Aided Verification*, New Brunswick, New Jersey, June 1990. Center for Discrete Mathematics and Theoretical Computer Science (DIMACS).

[Bur84] J. P. Burgess. Basic tense logoc. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic. Volume II: Extensions of Classical Logic*, pages 89–134. D. Reidel, 1984.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[CDK90] 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 *Fifteenth Colloquium on Trees in Algebra and Programming*, volume

431 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, May 1990. Springer-Verlag.

[CE81a] E. M. Clarke and E. A. Emerson. Characterizing properties of parallel programs as fixpoints. In *Seventh International Colloqium on Automata, Languages, and Programming*, volume 85 of *LNCS*, 1981.

[CE81b] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[CES86] 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, 1986.

[CGK89] E. M. Clarke, O. Grumberg, and R. P. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, volume 363 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1989.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[Cha87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. A. Plaice. Lustre, a declarative language for programming synchronous systems. In *14th ACM Symp. on Principles of Programming Languages*, January 1987.

[CLM89a] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.

[CLM89b] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *9th International Symposium on Hardware Description Languages and their Applications*, 1989.

[CM88] K. M. Chandy and J. Misra. *Parallel program design : a foundation.* Addison-Wesley, 1988.

[CMB91] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying temporal properties of sequential machines without building their state graphs. In E. M. Clarke and R. P. Kurshan, editors, *Computer Aided Verification, '90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1991.

[Cou91] O. Coudert. *SIAM: une boite à outils pour la preuve formelle de systèmes sequentiels.* PhD thesis, Ècole Nationale Supérieure des Télécommunications, 1991.

[DC86] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings, Pt. E*, 133(5):276–282, September 1986.

[Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. Technical Report 88-119, Carnegie Mellon University, Computer Science Dept, 1988.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the Second Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, June 1986.

[FB89] Allan L. Fisher and Randal E. Bryant. Performance of COSMOS on the IFIP workshop benchmarks. In *Proceedings of IMEC Conference*, 1989.

[FMK90] M. Fujita, Y. Matsunaga, and T Kakuda. Automatic and semi-automatic verification of switch-level circuits with temporal logic and binary decision diagrams. In *ICCAD*, pages 38–41, 1990.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Workshop on Computer Aided Verification*, 1990.

[GS] S. M. German and A. P. Sistla. Reasoning about systems with many processes. GTE Laboratories Inc., Waltham, Massachusetts.

[GS91] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In Robert Kurshan and Edmund M. Clarke, editors, *Computer-Aided Verification, Proceedings of the 1990 Workshop*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1991.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *LICS*, 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580,583, October 1969.

[KC90] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *1990 IEEE International Conference on Computer Design*, September 1990.

[KM89] R. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, 1989.

[Kro87] Fred Kroger. *Temporal Logic of Programs.* Springer-Verlag, 1987.

[Kur85] R. P. Kurshan. Modelling concurrent processes. In *Symp. in Applied Math. 31*, pages 45–57. 1985.

[Kur86] R. P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.

[Kur87] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19–39. Springer-Verlag, 1987.

[LN91] B. Lin and A. R. Newton. Efficient symbolic manipulation of equivalence relations and classes. In *IMEC-IFIP International Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles on Programming Languages*, 1985.

[LTN90] B. Lin, H. J. Touati, and A. R. Newton. Don't care minimization of multi-level sequential logic networks. In *ICCAD*, pages 414–417, 1990.

[Mar85] A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.

[MB59] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *The Annals of the Computation Laboratory of*

Harvard University. Volume XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I, pages 204–243. Harvard University Press, 1959.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mil83] G. J. Milne. Circal, calculus for circuit descriptions. *Integration*, 1:121–160, 1983.

[MO81] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations.* 1981.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. 1981.

[MS91] K.L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessors*, 1991.

[NH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(83), 1984.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[Par74] David Park. Finiteness is mu-ineffable. Theory of Computation Report No. 3, The University of Warwick, 1974.

[PL89] D. K. Probst and H. F. Li. Abstract specification, composition, and proof of correctness of delay-insensitive circuits and systems. Technical report, Concordia University, Dept. of Computer Science, 1989.

[PL90] D. K. Probst and H. F. Li. Using partial order semantics to avoid the state explosion problem in asynchronous systems. In *Workshop on Computer Aided Verification*, 1990.

[Pnu77] A. Pnueli. The temporal semantics of concurrent programs. In *18th Symposium on Foundations of Computer Science*, 1977.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Lecture Notes in Computer Science*, volume 224, pages 510–584. Springer-Verlag, 1986.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, 1971.

[Sac77] E. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, 1977.

[Sei80a] C. L. Seitz. Ideas about arbiters. *Lambda*, 10(14), 1980.

[Sei80b] C. L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, pages 151–165. Springer-Verlag, 1989.

[Smu68] R. M. Smullyan. *First Order Logic.* Springer-Verlag, 1968.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[TBK91] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. In *IMEC-IFIP International Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.

[Tho84] R. H. Thomason. Combinations of tense and modality. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic. Volume II: Extensions of Classical Logic*, pages 89–134. D. Reidel, 1984.

[TSL$^+$90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *ICCAD*, pages 130–133, 1990.

[Val89] A. Valmari. Stubborn sets for reduced state space generation. In *10th Int. Conf. on Application and Theory of Petri Nets*, 1989.

[Val90] A. Valmari. A stubborn attack on the state explosion problem. In *Workshop on Computer Aided Verification*, 1990.

[vdS83] Jan L. A. van de Snepscheut. *Trace Theory and VLSI design.* PhD thesis, Department of Computer Science, Eindhoven University of Technology, October 1983.

[WL89] P. Wolper and V Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, pages 68–80. Springer-Verlag, 1989.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[YTK91] Tomohiro Yoneda, Yoshihiro Tohma, and Yutaka Kondo. Acceleration of timing verification method based on time Petri nets. *Systems and Computers in Japan*, 22(12):37–52, 1991.