Verification Tools for Finite-State Concurrent Systems [⋆]

E. Clarke[1], O. Grumberg[2], and D. Long[3]

[1] Carnegie Mellon, Pittsburgh
[2] The Technion, Haifa
[3] AT&T Bell Labs, Murray Hill

ABSTRACT: Temporal logic model checking is an automatic technique for verifying finite-state concurrent systems. Specifications are expressed in a propositional temporal logic, and the concurrent system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specification. When the technique was first developed ten years ago, it was only possible to handle concurrent systems with a few thousand states. In the last few years, however, the size of the concurrent systems that can be handled has increased dramatically. By representing transition relations and sets of states implicitly using *binary decision diagrams*, it is now possible to check concurrent systems with more than 10^{120} states. In this paper we describe in detail how the new implementation works and give realistic examples to illustrate its power. We also discuss a number of directions for future research. The necessary background information on binary decision diagrams, temporal logic, and model checking has been included in order to make the exposition as self-contained as possible.

Keywords: automatic verification, temporal logic, model checking, binary decision diagrams

1 Introduction

Finite-state concurrent systems arise naturally in several areas of computer science, particularly in the design of digital circuits and communication protocols. Logical errors found late in the design phase of these systems are an extremely important problem for both circuit designers and programmers. Such errors can delay getting a new product on the market or cause the failure of some critical device that is already in use. The most widely used verification technique is based on extensive simulation and can easily

[⋆] This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the National Science foundation under Grant No. CCR-9217549 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the U.S. government.

miss significant errors when the number possible states of the circuit or protocol is very large. Although there has been considerable research on the use of theorem provers, term rewriting systems and proof checkers for verification, these techniques are time consuming and often require a great deal of manual intervention. During the past ten years, researchers at Carnegie Mellon University have developed an alternative approach to verification called *temporal logic model checking* [26, 27]. In this approach specifications are expressed in a propositional temporal logic, and circuit designs and protocols are modeled as state-transition systems. An efficient search procedure is used to determine automatically if the specifications are satisfied by the transition systems.

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

The main disadvantage of this technique is the state explosion which can occur if the system being verified has many components that can make transitions in parallel. Because of this problem, many researchers in formal verification predicted that model checking would never be practical for large circuits and protocols. Recently, however, the size of the transition systems that can be verified by model checking techniques has increased dramatically. The initial breakthrough was made in the fall of 1987 by McMillan, who was then a graduate student at Carnegie Mellon. He realized that using an explicit representation for transition relations severely limited the size of the circuits and protocols that could verified. He argued that larger systems could be handled if transition relations were represented implicitly with *ordered binary decision diagrams* (OBDDs) [14]. By using the original model checking algorithm with the new representation for transition relations, he was able to verify some examples that had more than 10^{20} states [22, 61]. He made this observation independently of the work by Coudert, et. al. [33] and Pixley [65, 66, 67] on using OBDDs to check equivalence of deterministic finite-state machines. Since then, various refinements of the OBDD-based techniques by other researchers at Carnegie Mellon have pushed the state count up to more than 10^{120} [19].

1.1 Temporal Logic model checking

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. They were originally developed by philosophers for investigating the way that time is used in natural language arguments [49]. Although a number of different temporal logics have been studied, most have an operator like $\mathbf{G}\,f$ that is true in the present if f is always true in the future (*i.e.*, if f is globally true). To assert that two events e_1 and e_2 never occur at the same time, one would write $\mathbf{G}(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure. This classification may occasionally be misleading since some temporal logics combine both linear-time and branching-time operators. Instead, we will adopt the approach used in [41] that permits both types of logics to be treated within a single semantical framework. In this paper the meaning of a temporal logic formula will always be determined with respect to a labelled state-transition graph; for historical reasons such structures are called *Kripke models* [49].

Pnueli [68] was the first to use temporal logic for reasoning about the concurrent programs. His approach involved proving properties of the program under consideration from a set of axioms that described the behavior of the individual statements in the program. The method was extended to sequential circuits by Bochmann [7] and Owicki and Malachi [58]. Since proofs were constructed by hand, the technique was often difficult to use in practice. The introduction of temporal logic model checking algorithms in the early 1980's allowed this type of reasoning to be automated. Since checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently. The first algorithm was developed by Clarke and Emerson in [26]. Their algorithm was polynomial in both the size of the model determined by the program under consideration and in the length of its specification in temporal logic. They also showed how *fairness* could be handled without changing the complexity of the algorithm. This was an important step since the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the assumption that each process makes progress infinitely often.

At roughly the same time Quielle and Sifakis [70] gave a model checking algorithm for a similar branching-time logic, but they did not analyze its complexity or show how to handle an interesting notion of fairness. Later Clarke, Emerson, and Sistla [27] devised an improved algorithm that was linear in the product of the length of the formula and in the size of the global state graph. Clarke and Sistla [74] also analyzed the model checking problem for a variety of other temporal logics and showed, in particular, that for linear temporal logic the problem was PSPACE complete.

A number of papers demonstrated how the temporal logic model checking procedure could be used for verifying network protocols and sequential circuits ([10], [11], [12], [13], [27], [38], [63]). In the case of sequential circuits two approaches were used for obtaining state-transition graphs to analyze. The first approach extracted a state graph directly from the circuit under an appropriate timing model of circuit behavior. The second approach obtained a state-transition graph by compilation from a high level representation of the circuit in a Pascal-like programming language. Early model checking systems were able to check state-transition graphs with between 10^4 and 10^5 states at a rate of about 100 states per second for formulas. In spite of these limitations, model checking systems were used successfully to find previously unknown errors in several published circuit designs.

Alternative techniques for verifying concurrent systems were proposed by a number of other researchers. The approach developed by Kurshan [48, 52] was based on checking *inclusion* between two automata. The first machine represented the system that was being verified; the second represented its specification. Automata on infinite tapes (ω-automata) were used in order to handle fairness. Pnueli and Lichtenstein [56] reanalyzed the complexity of checking linear-time formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph. Based on this observation, they argued that the high complexity of linear-time model checking might still be acceptable for short formulas. Emerson and Lei [43] extended their result to show that formulas of the logic CTL*, which combines both branching-time and linear-time operators, could be checked with essentially the same complexity as formulas of linear temporal logic. Vardi and Wolper [79] showed how the model checking problem could be formulated in terms of automata, thus relating the model checking approach to the work of Kurshan.

1.2 New implementations

In the original implementation of the model checking algorithm, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. Recent implementations [22, 61] use the same basic algorithm; however, transition relations are represented implicitly by *ordered binary decision diagrams* (OBDDs) [14]. OBDDs provide a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Because this representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states—many orders of magnitude larger than could be handled by the original algorithm.

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

The model checking system that McMillan developed as part of his Ph.D. thesis is called SMV [61]. It is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system from a program in the SMV language and uses a OBDD-based search algorithm to determine whether the system satisfies its specifications. If the transition system does not satisfy some specification, the verifier will produce an execution trace that shows why the specification is false. The SMV system has been distributed widely, and a large number of examples have now been verified with it. These examples provide convincing evidence that SMV can be used to debug real industrial designs.

Perhaps, the most impressive example that has been verified by model checking techniques is the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991). Although development of the Futurebus+ cache coherence protocol began in 1988, all previous attempts to validate the protocol were based entirely on informal techniques. In the summer of 1992 researchers at Carnegie Mellon [29] constructed a precise model of the protocol in SMV language and then used SMV to show that the resulting transition system satisfied a formal specification of cache coherence. They were able to find a number of previously undetected errors and potential errors in the design of the protocol. This appears to be the first time that an automatic verification tool has been used to find errors in an IEEE standard.

1.3 Related verification techniques

A number of other researchers have independently discovered that OBDDs can be used to represent large state-transition systems. Coudert, Berthet, and Madre [33] have developed an algorithm for showing equivalence between two deterministic finite-state automata by performing a breadth first search of the state space of the product automata. They use OBDDs to represent the transition functions of the two automata in their algorithm. Similar algorithms have been developed by Pixley [65, 66, 67]. In addition, several groups including Bose and Fisher [8], Pixley [65], and Coudert, et. al. [34] have experimented with model checking algorithms that use OBDDs. Although the results of McMillan's experiments [21, 22] were not published until the summer of 1990, his work is referenced by Bose and Fisher in their 1989 paper [8].

Recently, Bryant, Seger and Beatty [4, 17] have developed an algorithm based on symbolic simulation for model checking in a restricted linear time logic. Specifications consist of precondition–postcondition pairs expressed in the logic. The precondition is used to restrict inputs and initial states of the circuit; the postcondition gives the property that the user wishes to check. Formulas in the logic have the form

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

The syntax of the formulas is highly restricted compared to most other temporal logics used for specifying programs and circuits. In particular, the only logical operator that is allowed is conjunction, and the only temporal operator is *next time* ($\mathbf{X}$). By limiting the class of formulas that can be handled, it is possible to check certain properties very efficiently. In many cases, however, these restrictions can be a disadvantage, since the number of time units that a formula can "look ahead in the future" is bounded by the maximum nesting of $\mathbf{X}$ operators.

It is difficult to compare the performance of the various symbolic verification methods. Probably, the best method is to study how the CPU time required for verification grows asymptotically with larger and larger instances of the circuit or protocol. In most of the example circuits considered in [19, 20], this growth rate is a small polynomial in the number of components of the circuit. Of the other groups mentioned above, only Bryant, Beatty and Seger [4] have demonstrated good asymptotic performance on a nontrivial class of circuits. Berthet, Coudert and Madre [6] obtained verification times that were sublinear in the number of states in the system, but these times were still exponential in the number of components.

1.4 Outline of paper

Our paper is organized as follows: The properties of OBDDs that are needed to understand the paper are given in Section 2. The next section shows how relations over a finite domain can be encoded using OBDDs. Section 4 describes the logics that are used in this paper, and Section 5 discusses some of the properties of predicate transformers that are needed in model checking. The basic model checking algorithm for branching-time temporal logic is given in Section 6. The next two sections describe extensions of the basic algorithm. Section 7 shows how fairness constraints can be handled, and Section 8 describes how counterexamples and witnessnesses are generated. A model checking algorithm for linear-time temporal logic is presented in Section 9, and the following section shows how model checking techniques can be used to test inclusion between ω-automata. Section 11 gives a brief overview of the SMV model checking system, and Section 12 shows how SMV was used to find errors in the IEEE Futurebus standard. The paper concludes in Section 13 with some directions for future research.

2 Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are a canonical form representation for boolean formulas [14]. They are often substantially more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form, and they can be manipulated very efficiently. Hence, they have become widely used for a variety of CAD applications, including symbolic simulation, verification of combinational logic and, more recently, verification of sequential circuit designs.

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

Binary decision trees do not provide a very concise representation for boolean functions. In fact, they are essentially the same size as truth tables. Fortunately, there is usually a lot of redundancy in such trees. For example, in the tree of Figure 1 there are

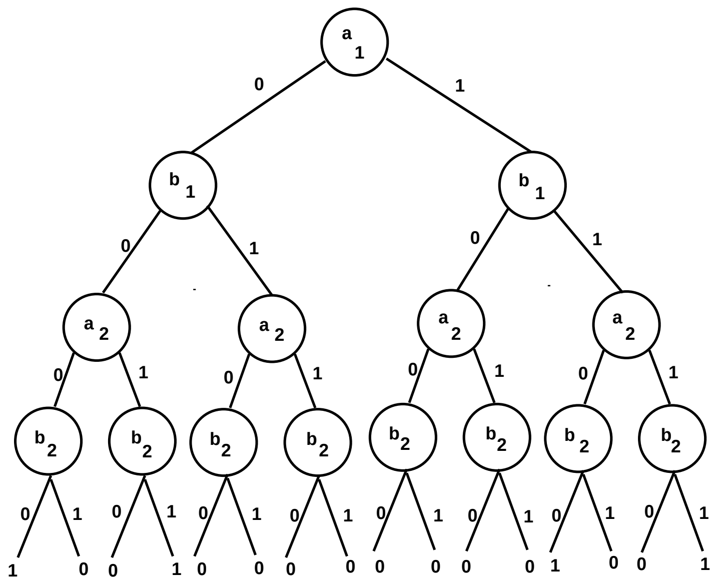


Fig. 1. Binary decision tree for two-bit comparator

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

1. If v is a terminal vertex:
   (a) If $value(v) = 1$ then $f_v(x_1, \ldots, x_n) = 1$.
   (b) If $value(v) = 0$ then $f_v(x_1, \ldots, x_n) = 0$.
2. If v is a nonterminal vertex with $var(v) = x_i$ then f_v is the function

$$f_v(x_1, \ldots, x_n) = \bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) \; + \; x_i \cdot f_{high(v)}(x_1, \ldots, x_n)$$

In practical applications it is desirable to have a *canonical representation* for boolean functions. Such a representation must have the property that two boolean functions are logically equivalent if and only if they have isomorphic representations. This property simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not. Two binary decision diagrams are *isomorphic* if there exists a one-to-one and onto function h that maps terminals of one to terminals of the other and nonterminals of one to nonterminals of the other, such that for every terminal vertex

v, $value(v) = value(h(v))$ and for every nonterminal vertex v, $var(v) = var(h(v))$, $h(low(v)) = low(h(v))$, and $h(high(v)) = high(h(v))$.

Bryant [14] showed how to obtain a canonical representation for boolean functions by placing two restrictions on binary decision diagrams. First, the variables should appear in the same order along each path from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is achieved by imposing a total ordering $<$ on the variables that label the vertices in the binary decision diagram and requiring that for any vertex u in the diagram, if u has a nonterminal successor v, then $var(u) < var(v)$. The second requirement is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram:

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

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

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

Starting with a binary decision diagram satisfying the ordering property, the canonical form is obtained by applying the transformation rules until the size of the diagram can no longer be reduced. Bryant shows how this can be done in a bottom-up manner by a procedure called *Reduce* in time which is linear in the size of the original binary decision diagram [14]. The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner. For example, if we use the ordering $a_1 < b_1 < a_2 < b_2$ for the two-bit comparator function, we obtain the OBDD shown in Figure 2. If OBDDs

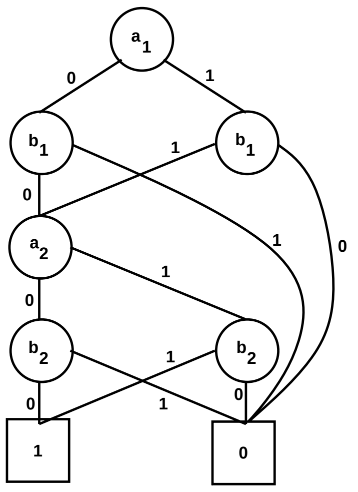


Fig. 2. OBDD for two-bit comparator

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

The size of an OBDD can depend critically on the variable ordering. For example, if we use the variable ordering $a_1 < a_2 < b_1 < b_2$ for the bit-comparator function, we get the OBDD shown in Figure 3. Note that this OBDD has 11 vertices while the OBDD shown in Figure 2 has only 8 vertices. In general, for n-bit comparator, if we choose the ordering $a_1 < b_1 < \ldots < a_n < b_n$, then the number of OBDD vertices will be $3n + 2$. On the other hand, if we choose the ordering $a_1 < \ldots < a_n < b_1 \ldots < b_n$, then the number of OBDD vertices is $3 \cdot 2^n - 1$. Finding an optimal ordering can be shown to be NP-complete in general. Moreover, there are boolean functions that have exponential size OBDDs for any variable ordering. One example is the boolean function for the middle output (or n-th output) of a combinational circuit to multiply two n bit integers [16, 15].

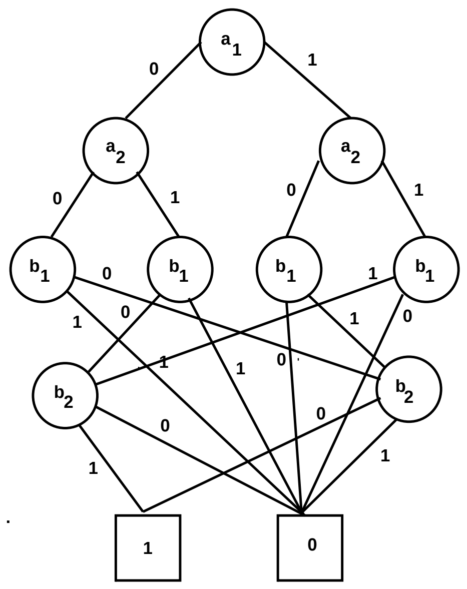


Fig. 3. OBDD for two-bit comparator

Several heuristics have been developed for finding a good variable ordering when such an ordering exists. If the boolean function is given by a combinational circuit, then heuristics based on a depth-first traversal of the circuit diagram generally give good results [44, 59]. The intuition for these heuristics comes from the observation that OBDDs tend to be small when related variables are close together in the ordering. The variables appearing in a subcircuit are related in that they determine the subcircuit's output. Hence, these variables should usually be grouped together in the ordering. This may be accomplished by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit diagram. A technique, called *dynamic reordering* [71], appears to be useful in those situations where no obvious ordering heuristics apply . When this technique is used, the OBDD package internally reorders the variables periodically in order to reduce the total number of vertices in use. The reordering method is designed to be fast rather than to find an optimal ordering.

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

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

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

All 16 two-argument logical operations can be implemented efficiently on boolean functions that are represented as OBDDs. In fact, the complexity of these operations is linear in the size of the argument OBDDs. The key idea for efficient implementation of these operations is the *Shannon expansion*

$$f = \bar{x} \cdot f\mid_{x \leftarrow 0} + x \cdot f\mid_{x \leftarrow 1} .$$

Bryant [14] gives a uniform algorithm called *Apply* for computing all 16 logical operations. Below we briefly explain how *Apply* works. Let $\star$ be an arbitrary two argument logical operation, and let f and f' be two boolean functions. To simplify the explanation of the algorithm we introduce the following notation:

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

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

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

$$f \star f' = \bar{x} \cdot (f\mid_{x \leftarrow 0} \star f'\mid_{x \leftarrow 0}) + x \cdot (f\mid_{x \leftarrow 1} \star f'\mid_{x \leftarrow 1})$$

to break the problem into two subproblems. The subproblems are solved recursively. The root of the resulting OBDD will be v with $var(v) = x$. $Low(v)$ will be the OBDD for $(f\mid_{x \leftarrow 0} \star f'\mid_{x \leftarrow 0})$ and $high(v)$ will be the OBDD for $(f\mid_{x \leftarrow 1} \star f'\mid_{x \leftarrow 1})$.
– If $x < x'$, then $f'\mid_{x \leftarrow 0} = f'\mid_{x \leftarrow 1} = f'$ since f' does not depend on x. In this case the Shannon Expansion simplifies to

$$f \star f' = \bar{x} \cdot (f\mid_{x \leftarrow 0} \star f') + x \cdot (f\mid_{x \leftarrow 1} \star f')$$

and the OBDD for $f \star f'$ is computed recursively as in the second case.
– If $x' < x$, then the required computation is similar to the previous case.

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

Several extensions have been developed to decrease the space requirements of Bryant's original OBDD representation for boolean functions [9]. A single multi-rooted graph can be used to represent a collection of boolean functions that share subgraphs. The same variable ordering is used for all of the formulas in the collection. As in the case of standard OBDDs, the graph contains no isomorphic subgraphs or redundant vertices. If this extension is used then two functions in the collection are identical if and only if they have the same root. Consequently, checking whether two functions are equal can be implemented in constant time. Another useful extension is adding labels to the arcs in the graph to denote boolean negation. This makes it unnecessary to use different subgraphs to represent a formula and its negation. Modern OBDD packages permit graphs with hundreds of thousands of vertices to be manipulated efficiently.

OBDDs can also be viewed as a form of deterministic finite automata [31]. An n-argument boolean function can be identified with the set of strings in $\{0,1\}^n$ that evaluate to 1. Since this is a finite language and all finite languages are regular, there is a minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original boolean function. Logical operations on boolean functions can be implemented by set operations on the languages accepted by the finite automata. For example, AND corresponds to set intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages. The standard OBDD operations can be viewed as analogs of these constructions.

3 Representing relations with OBDDs

OBDDs are extremely useful for obtaining concise representations of relations over finite domains [22, 61]. If R is n-ary relation over $\{0,1\}$ then R can be represented by the OBDD for its *characteristic function*

$$f_R(x_1,\ldots,x_n) = 1 \text{ iff } R(x_1,\ldots,x_n).$$

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

$$R'(\bar{x}_1,\ldots,\bar{x}_n) = R(\phi(\bar{x}_1),\ldots,\phi(\bar{x}_n))$$

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

In order to construct complex relations it is convenient to extend propositional logic to permit quantification over boolean variables. The resulting logic is called QBF (*Quantified Boolean Formulas*) [1, 45] and has the following syntax. Given a set $V = \{v_1, \ldots, v_n\}$ of propositional variables, $QBF(V)$ is the smallest set of formulas such that

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

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

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

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

- $\sigma \models v$ iff $\sigma(v) = 1$,
- $\sigma \models \neg f$ iff $\sigma \not\models f$,
- $\sigma \models f \vee g$ iff $\sigma \models f$ or $\sigma \models g$,
- $\sigma \models f \wedge g$ iff $\sigma \models f$ and $\sigma \models g$,
- $\sigma \models \exists v f$ iff $\sigma\langle v \leftarrow 0\rangle \models f$ or $\sigma\langle v \leftarrow 1\rangle \models f$, and
- $\sigma \models \forall v f$ iff $\sigma\langle v \leftarrow 0\rangle \models f$ and $\sigma\langle v \leftarrow 1\rangle \models f$,

QBF formulas have the same expressive power as ordinary propositional formulas; however, they are sometimes much more concise. Every QBF formula determines an n-ary boolean relation on the set V which consists of those truth assignments for the variables in V that make the formula true. We will identify each QBF formula with the boolean relation that it determines. In the previous section we showed how to associate an OBDD with each formula of propositional logic. In principle, it is easy to construct OBDDs for $\exists v f$ and $\forall v f$ when f is given as an OBDD.

- $\exists x f = f\,|_{x \leftarrow 0} + f\,|_{x \leftarrow 1}$
- $\forall x f = f\,|_{x \leftarrow 0} \cdot f\,|_{x \leftarrow 1}$

In practice, however, special algorithms are needed to handle quantifiers efficiently [20]. In this paper quantifiers occur most frequently in *relational products* which have the following form

$$\exists \bar{v}\big[f(\bar{v}) \wedge g(\bar{v})\big].$$

We will restrict our attention to this case. In Figure 4, we give an algorithm *RelProd* that performs this computation in one pass over the BDDs $f(\bar{v})$ and $g(\bar{v})$. This is important in practice since the relational product is computed without ever constructing the BDD for

$$f(\bar{v}) \wedge g(\bar{v}),$$

which is often fairly large. Like many OBDD algorithms, *RelProd* uses a result cache. In

```
function $RelProd(f, g\colon OBDD,\ E\colon set\ of\ variables)\colon OBDD$

if $f = false \vee g = false$
    return $false$
else if $f = true \wedge g = true$
    return $true$
else if $(f, g, E, h)$ is in the result cache
    return $h$
else
    let $x$ be the top variable of $f$
    let $y$ be the top variable of $g$
    let $z$ be the topmost of $x$ and $y$
    $h_0 := RelProd(f|_{z=0}, g|_{z=0}, E)$
    $h_1 := RelProd(f|_{z=1}, g|_{z=1}, E)$
    if $z \in E$
        $h := Or(h_0, h_1)$
            /* OBDD for $h_0 \vee h_1$ */
    else
        $h := IfThenElse(z, h_1, h_0)$
            /* OBDD for $(z \wedge h_1) \vee (\neg z \wedge h_0)$ */
    endif
    insert $(f, g, E, h)$ in the result cache
    return $h$
endif
```

Fig. 4. Relational product algorithm

this case, entries in the cache are of the form (f, g, E, h), where E is a set of variables that are quantified out and f, g and h are OBDDs. If such an entry is in the cache, it means that a previous call to $RelProd(f, g, E)$ returned h as its result.

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

4 Computation Tree Logics

In this paper finite-state systems are modeled by labeled state-transition graphs, called *Kripke Structures* [49]. If some state is designated as the *initial state*, then the Kripke structure can be unwound into an infinite tree with that state as the root, as illustrated in Figure 5. Since paths in the tree represent possible computations of the program, we will refer to the infinite tree obtained in this manner as the computation tree of the program. Temporal logics may differ according to how they handle branching in

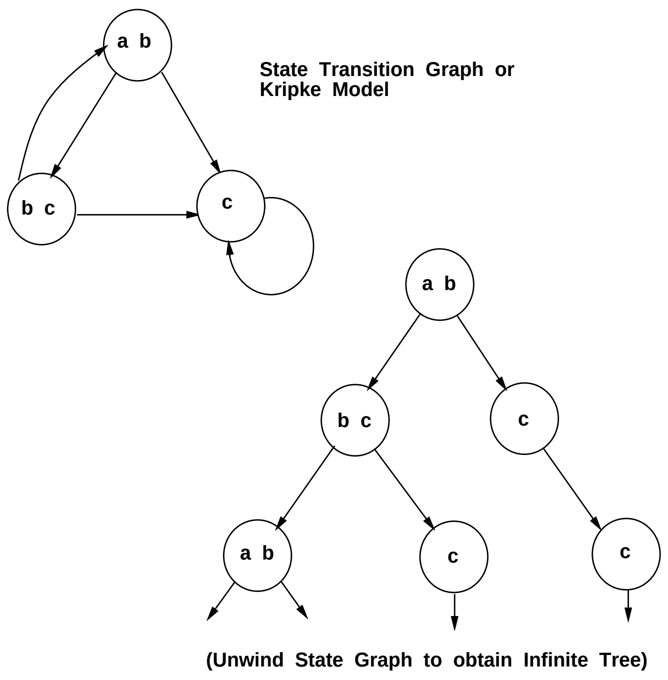

a b
b c
c
State Transition Graph or Kripke Model
a b
b c
c
a b
c
c
(Unwind State Graph to obtain Infinite Tree)

Fig. 5. Basic Model of Computation

the underlying computation tree. In a linear temporal logic, operators are provided for describing events along a single computation path. In a branching-time logic the temporal operators quantify over the paths that are possible from a given state. The computation tree logic CTL* [26, 27, 41] combines both branching-time and linear-time operators; a path quantifier, either **A** ("for all computation paths") or **E** ("for some computation paths") can prefix an assertion composed of arbitrary combinations of the usual linear-time operators **G** ("always"), **F** ("sometimes"), **X** ("nexttime"), and **U** ("until"). The remainder of this section gives a precise description of the syntax and semantics of these logics.

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

– If $p \in AP$, then p is a state formula.
– If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.

– If f is a path formula, then $\mathbf{E}(f)$ is a state formula.

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

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

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

We define the semantics of CTL* with respect to a Kripke structure $M = \langle S, R, L\rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation, which must be *total* (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \rightarrow \mathcal{P}(AP)$ is a function that labels each state with a set of atomic propositions true in that state. Unless otherwise stated, all of our results apply only to *finite* Kripke structures.

A *path in M* is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. We use π^i to denote the *suffix* of π starting at s_i. If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in Kripke structure M. When the Kripke structure M is clear from context, we will usually omit it. The relation $\models$ is defined inductively as follows (assuming that f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

1. $s \models p$	$\Leftrightarrow p \in L(s)$.
2. $s \models \neg f_1$	$\Leftrightarrow s \not\models f_1$.
3. $s \models f_1 \vee f_2$	$\Leftrightarrow s \models f_1$ or $s \models f_2$.
4. $s \models \mathbf{E}(g_1)$	$\Leftrightarrow$ there exists a path π starting with s such that $\pi \models g_1$.
5. $\pi \models f_1$	$\Leftrightarrow s$ is the first state of π and $s \models f_1$.
6. $\pi \models \neg g_1$	$\Leftrightarrow \pi \not\models g_1$.
7. $\pi \models g_1 \vee g_2$	$\Leftrightarrow \pi \models g_1$ or $\pi \models g_2$.
8. $\pi \models \mathbf{X}\, g_1$	$\Leftrightarrow \pi^1 \models g_1$.
9. $\pi \models g_1\, \mathbf{U}\, g_2$	$\Leftrightarrow$ there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k$, $\pi^j \models g_1$.

The following abbreviations are used in writing CTL* formulas:

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

CTL [5, 26] is a restricted subset of CTL* that permits only branching-time operators—each of the linear-time operators $\mathbf{G}$, $\mathbf{F}$, $\mathbf{X}$, and $\mathbf{U}$ must be immediately preceded by a path quantifier. More precisely, CTL is the subset of CTL* that is obtained if the following two rules are used to specify the syntax of path formulas.

– If f and g are state formulas, then $\mathbf{X}\, f$ and $f\, \mathbf{U}\, g$ are path formulas.
– If f is a path formula, then so is $\neg f$.

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

– If $p \in AP$, then p is a path formula.

– If f and g are path formulas, then $\neg f$, $f \vee g$, $\mathbf{X}\, f$, and $f\, \mathbf{U}\, g$ are path formulas.

It can be shown [24, 41, 55] that the three logics discussed in this section have different expressive powers. For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}\, p)$. Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}\, p)$. The disjunction of these two formulas $\mathbf{A}(\mathbf{FG}\, p) \vee \mathbf{AG}(\mathbf{EF}\, p)$ is a CTL* formula that is not expressible in either CTL or LTL.

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

– **AX** and **EX**,
– **AG** and **EG**,
– **AF** and **EF**,
– **AU** and **EU**

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

– $\mathbf{AX}\, f = \neg\, \mathbf{EX}(\neg f)$
– $\mathbf{AG}\, f = \neg\, \mathbf{EF}(\neg f)$
– $\mathbf{AF}\, f = \neg\, \mathbf{EG}(\neg f)$
– $\mathbf{EF}\, f = \mathbf{E}[true\, \mathbf{U}\, f]$
– $\mathbf{A}[f\, \mathbf{U}\, g] \equiv \neg\, \mathbf{E}[\neg g\, \mathbf{U} \neg f \wedge \neg g] \wedge \neg\, \mathbf{EG} \neg g$

The four operators that are used most widely are illustrated in Figure 6. Each computation tree has the state s_0 as its root.

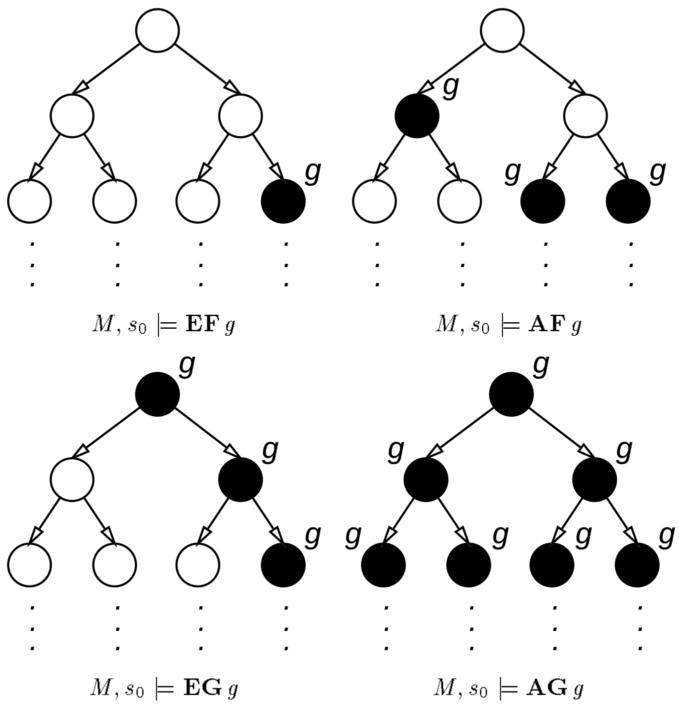
Finally, some typical CTL formulas that might arise in verifying a finite state concurrent program are given below:

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

5 Fixpoint characterization

Let $M = (S, R, L)$ be an arbitrary finite Kripke structure. We use $Pred(S)$ to denote the lattice of predicates over S where each predicate is identified with the set of states in S that make it true and the ordering is set inclusion. Thus, the least element in the lattice is the empty set, denoted by $False$, and the greatest element in the lattice is the set of all states, denoted by $True$. A functional F that maps $Pred(S)$ to $Pred(S)$ will be called a *predicate transformer*. Let $\tau : Pred(S) \longrightarrow Pred(S)$ be such a functional, then

1. τ is *monotonic* provided that $P \subseteq Q$ implies $\tau[P] \subseteq \tau[Q]$;
2. τ is $\cup$-*continuous* provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau[\cup_i P_i] = \cup_i \tau[P_i]$;



g
g
g
g
$M, s_0 \models \mathbf{EF}\, g$
$M, s_0 \models \mathbf{AF}\, g$
g
g
g
g
g
g
g
g
g
g
$M, s_0 \models \mathbf{EG}\, g$
$M, s_0 \models \mathbf{AG}\, g$

Fig. 6. Basic CTL Operators

3. τ is $\cap$-*continuous* provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau[\cap_i P_i] = \cap_i \tau[P_i]$.

A monotonic predicate transformer τ on $Pred(S)$ always has a least fixpoint, $\mathbf{lfp}\, Z\, [\tau(Z)]$, and a greatest fixpoint, $\mathbf{gfp}\, Z\, [\tau(Z)]$ (see Tarski [76]): $\mathbf{lfp}\, Z\, [\tau(Z)] = \cap\{Z \mid \tau(Z) = Z\}$ whenever τ is monotonic, and $\mathbf{lfp}\, Z\, [\tau(Z)] = \cup_i \tau^i(False)$ whenever τ is also $\cup$-continuous; $\mathbf{gfp}\, Z\, [\tau(Z)] = \cup\{Z \mid \tau(Z) = Z\}$ whenever τ is monotonic, and $\mathbf{gfp}\, Z\, [\tau(Z)] = \cap_i \tau^i(True)$ whenever τ is also $\cap$-continuous.

The following lemmas are useful in working with predicate transformers defined on finite Kripke structures.

Lemma 1. *Let M be a finite Kripke structure and let τ be a functional over $Pred(S)$. If τ is monotonic then τ is also $\cup$-continuous and $\cap$-continuous.*

Lemma 2. *If τ is monotonic, then for every i, $\tau^i(False) \subseteq \tau^{i+1}(False)$ and $\tau^i(True) \supseteq \tau^{i+1}(True)$.*

Lemma 3. *If τ is monotonic and M is finite, then there is an integer i_0 such that for every $j \geq i_0$, $\tau^j(False) = \tau^{i_0}(False)$. Similarly, there is some j_0 such that for every $j \geq j_0$, $\tau^j(True) = \tau^{j_0}(True)$.*

Lemma 4. *If τ is monotonic and M is finite, then there is an integer i_0 such that $\mathbf{lfp}\, Z\left[\tau(Z)\right] = \tau^{i_0}(False)$. Similarly, there is an integer j_0 such that $\mathbf{gfp}\, Z\left[\tau(Z)\right] = \tau^{j_0}(True)$.*

As a consequence of the preceding lemmas, if τ is monotonic, its least fixpoint can be computed by the program in Figure 7. The invariant for the while loop in the body

```
function Lfp(Tau : PredicateTransformer): Predicate
begin
    $Q := False$;
    $Q' := Tau(Q)$;
    while ($Q \neq Q'$) do
    begin
        $Q := Q'$;
        $Q' := Tau(Q')$
    end;
    return($Q$)
end
```

Fig. 7. Procedure for computing least fixpoints.

of the procedure is given by the assertion

$$(Q' = \tau[Q]) \wedge (Q' \subseteq \mathbf{lfp}\, Z\left[\tau(Z)\right])$$

It is easy to see that at the beginning of the i-th iteration of the loop, $Q = \tau^{i-1}(False)$ and $Q' = \tau^{i}(False)$. Lemma 2 implies that

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

Consequently, the maximum number of iterations before the while loop terminates is bounded by the number of elements in the set S. When the loop does terminate, we will have that $Q = \tau[Q]$ and that $Q \subseteq \mathbf{lfp}\, Z\left[\tau(Z)\right]$. It follows directly that $Q = \mathbf{lfp}\, Z\left[\tau(Z)\right]$ and that the value returned by the procedure is the required least fixpoint. The greatest fixpoint of τ may be computed in a similar manner by the program in Figure 8. Essentially the same argument can be used to show that the procedure terminates and that the value it returns is $\mathbf{gfp}\, Z\left[\tau(Z)\right]$.

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

- $\mathbf{A}[f_1\, \mathbf{U}\, f_2] = \mathbf{lfp}\, Z\left[f_2 \vee (f_1 \wedge \mathbf{AX}\, Z)\right]$
- $\mathbf{E}[f_1\, \mathbf{U}\, f_2] = \mathbf{lfp}\, Z\left[f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)\right]$
- $\mathbf{AF}\, f_1 = \mathbf{lfp}\, Z\left[f_1 \vee \mathbf{AX}\, Z\right]$
- $\mathbf{EF}\, f_1 = \mathbf{lfp}\, Z\left[f_1 \vee \mathbf{EX}\, Z\right]$

```
**function** $Gfp(Tau : PredicateTransformer) : Predicate$
**begin**
    $Q := True;$
    $Q' := Tau(Q);$
    **while** $(Q \neq Q')$ **do**
    **begin**
        $Q := Q';$
        $Q' := Tau(Q')$
    **end**;
    **return**$(Q)$
**end**
```

Fig. 8. Procedure for computing greatest fixpoints.

- $\mathbf{AG}\, f_1 = \mathbf{gfp}\, Z\,[f_1 \wedge \mathbf{AX}\, Z]$
- $\mathbf{EG}\, f_1 = \mathbf{gfp}\, Z\,[f_1 \wedge \mathbf{EX}\, Z]$

We will only prove the fixpoint characterizations for **EG** and **EU**. The fixpoint characterizations of the remaining CTL operators can be established in a similar manner.

Lemma 5. *$\tau(Z) = f_1 \wedge \mathbf{EX}\, Z$ is monotonic.*

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

Lemma 6. *Let $\tau(Z) = f_1 \wedge \mathbf{EX}\, Z$ and let $\tau^{i_0}(True)$ be the limit of the sequence $True \supseteq \tau(True) \supseteq \ldots$. For every $s \in S$, if $s \in \tau^{i_0}(True)$ then $s \models f_1$, and there is a state s' such that $(s, s') \in R$ and $s' \in \tau^{i_0}(True)$.*

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

Lemma 7. *$\mathbf{EG}\, f_1$ is a fixpoint of the functional $\tau(Z) = f_1 \wedge \mathbf{EX}\, Z$.*

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

Lemma 8. *$\mathbf{EG}\, f_1$ is the greatest fixpoint of the functional $\tau(Z) = f_1 \wedge \mathbf{EX}\, Z$.*

Proof. Since τ is monotonic, by lemma 1 it is also $\cap$-continuous. Therefore, in order to show that $\mathbf{EG}\, f_1$ is the greatest fixpoint of τ, it is sufficient to prove that $\mathbf{EG}\, f_1 = \cap_i \tau^i(True)$.

We first show that $\mathbf{EG}\, f_1 \subseteq \cap_i \tau^i(True)$. We establish this claim by applying induction on i to show that, for every i, $\mathbf{EG}\, f_1 \subseteq \tau^i(True)$. Clearly, $\mathbf{EG}\, f_1 \subseteq True$. Assume that $\mathbf{EG}\, f_1 \subseteq \tau^n(True)$. Since τ is monotonic, $\tau[\mathbf{EG}\, f_1] \subseteq \tau^{n+1}(True)$. By Lemma 7, $\tau[\mathbf{EG}\, f_1] = \mathbf{EG}\, f_1$. Hence, $\mathbf{EG}\, f_1 \subseteq \tau^{n+1}(True)$.

To show that $\cap_i \tau^i(True) \subseteq \mathbf{EG}\, f_1$, consider some state $s \in \cap_i \tau^i(True)$. This state is included in every $\tau^i(True)$. Hence, it is also in the fixpoint $\tau^{i_0}(True)$. By Lemma 6, s is the start of an infinite sequence of states in which each state is related to the previous one by the relation R. Furthermore, each state in the sequence satisfies f_1. Thus, $s \models \mathbf{EG}\, f_1$. $\square$

Lemma 9. $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ *is the least fixpoint of the functional* $\tau(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)$.

Proof. First we notice that $\tau(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)$ is monotonic. By Lemma 1, τ is therefore $\cup$-continuous. It is also straightforward to show that $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ is a fixpoint of $\tau(Z)$. We still need to prove that $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$ is the least fixpoint of $\tau(Z)$. For that, it is sufficient to show that $\mathbf{E}[f_1\ \mathbf{U}\ f_2] = \cup_i \tau^i(False)$. For the first direction, it is easy to prove by induction on i that for every i, $\tau^i(False) \subseteq \mathbf{E}[f_1\ \mathbf{U}\ f_2]$. Consequently, we have that $\cup_i \tau^i(False) \subseteq \mathbf{E}[f_1\ \mathbf{U}\ f_2]$.

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

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

Figure 9 shows how the set of states that satisfy $\mathbf{E}[p\ \mathbf{U}\ q]$ may be computed for a simple Kripke structure by using the procedure Lfp. In this case the functional τ is given by

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

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

6 Symbolic Model Checking

Model checking is the problem of finding the set of states in a state transition graph where a given CTL formula is true. There is a program called EMC (Extended Model Checker) that solves this problem using efficient graph-traversal techniques. If the model is represented as a state transition graph, the complexity of the algorithm is linear in the size of the graph and in the length of the formula. The algorithm is quite fast in

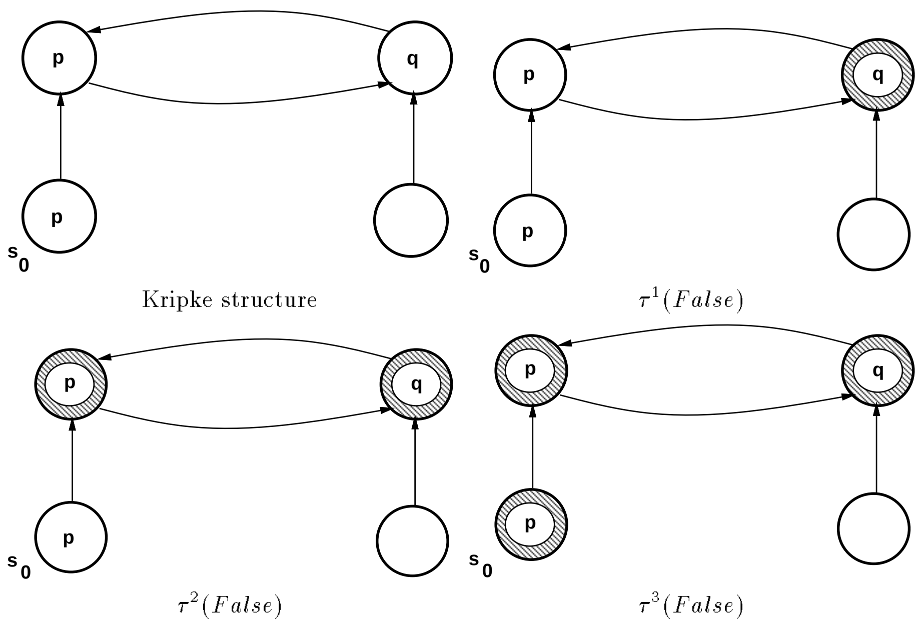


Fig. 9. Sequence of Approximations for $\mathbf{E}[p\ \mathbf{U}\ q]$

practice [26, 27]. However, an explosion in the size of the model may occur when the state transition graph is extracted from a finite state concurrent system that has many processes or components.

In this section, we describe a symbolic model checking algorithm for CTL which uses OBDDs to represent the state transition graph. Assume that the behavior of the concurrent system is determined by n boolean state variables $v_1, v_2, \ldots, v_n$. The transition relation $R(\bar{v}, \bar{v}')$ for the concurrent system will be given as a boolean formula in terms of two copies of the state variables: $\bar{v} = (v_1, \ldots, v_n)$ which represents the current state and $\bar{v}' = (v_1', \ldots, v_n')$ which represents the next state. The formula $R(\bar{v}, \bar{v}')$ is now converted to an OBDD. This usually results in a very concise representation of the transition relation.

The symbolic model checking algorithm is implemented by a procedure *Check* that takes the CTL formula to be checked as its argument and returns an OBDD that represents exactly those states of the system that satisfy the formula. Of course, the output of *Check* depends on the system being checked; this parameter is implicit in the discussion below. We define *Check* inductively over the structure of CTL formulas. If f is an atomic proposition v_i, then $Check(f)$ is simply the OBDD for v_i. Formulas of the form $\mathbf{EX}\, f$,

$\mathbf{E}[f\,\mathbf{U}\,g]$, and $\mathbf{EG}\,f$ are handled by the procedures:

$$\begin{array}{l} \mathit{Check}(\mathbf{EX}\,f) = \mathit{CheckEX}(\mathit{Check}(f)), \\ \mathit{Check}(\mathbf{E}[f\,\mathbf{U}\,g]) = \mathit{CheckEU}(\mathit{Check}(f), \mathit{Check}(g)), \\ \mathit{Check}(\mathbf{EG}\,f) = \mathit{CheckEG}(\mathit{Check}(f)). \end{array}$$

Notice that these intermediate procedures take boolean formulas as their arguments, while *Check* takes a CTL formula as its argument. The cases of CTL formulas of the form $f \vee g$ or $\neg f$ are handled using the standard algorithms for computing boolean connectives with OBDDs. Since $\mathbf{AX}\,f$, $\mathbf{A}[f\,\mathbf{U}\,g]$ and $\mathbf{AG}\,f$ can all be rewritten using just the above operators, this definition of *Check* covers all CTL formulas.

The procedure for *CheckEX* is straightforward since the formula $\mathbf{EX}\,f$ is true in a state if the state has a successor in which f is true.

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

If we have OBDDs for f and R, then we can compute an OBDD for

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

by using the techniques described in Section 3.

The procedure for *CheckEU* is based on the least fixpoint characterization for the CTL operator $\mathbf{EU}$ that is given in Section 5.

$$\mathit{CheckEU}(f(\bar{v}), g(\bar{v})) = \mathbf{lfp}\, Z(\bar{v}) \left[g(\bar{v}) \vee \left(f(\bar{v}) \wedge \mathit{CheckEX}(Z(\bar{v})) \right) \right].$$

In this case we use the function Lfp to compute a sequence of approximations

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

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

CheckEG is similar. In this case the procedure is based on the greatest fixpont characterization for the CTL operator $\mathbf{EG}$ that is given in Section 5.

$$\mathit{CheckEG}(f(\bar{v})) = \mathbf{gfp}\, Z(\bar{v}) \left[f(\bar{v}) \wedge \mathit{CheckEX}(Z(\bar{v})) \right].$$

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

7 Fairness Constraints

Next, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along fair computation paths. For example, if we are verifying an asynchronous circuit with an arbiter, we may wish to consider only those executions in which the arbiter does not ignore one of its request inputs forever. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of CTL slightly. A *fairness constraint* can be an arbitrary set of states, usually described by a formula of the logic. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The path quantifiers in CTL formulas are then restricted to fair paths. In the remainder of this section we describe how to modify the algorithm above to handle fairness constraints. We assume the fairness constraints are given by a set of CTL formulas $H = \{h_1, \ldots, h_n\}$. We define a new procedure *CheckFair* for checking CTL formulas relative to the fairness constraints in H. We do this by giving definitions for new intermediate procedures *CheckFairEX*, *CheckFairEU*, and *CheckFairEG* which correspond to the intermediate procedures used to define *Check*.

Consider the formula $\mathbf{EG}\, f$ given fairness constraints H. The formula means that there exists a path beginning with the current state on which f holds globally (invariantly) and each formula in H holds infinitely often on the path. The set of such states S is the largest set with the following two properties:

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

It is easy to show that if these conditions hold, each state in the set is the beginning of an infinite computation path on which f is always true, and for which every formula in H holds infinitely often. Thus, the procedure $\mathit{CheckFairEG}(f(\bar{v}))$ will compute the greatest fixpoint

$$\mathbf{gfp}\, Z(\bar{v}) \left[f(\bar{v}) \wedge \bigwedge_{k=1}^{n} \mathit{CheckEX}(\mathit{CheckEU}(f(\bar{v}), Z(\bar{v}) \wedge \mathit{Check}(h_k))) \right].$$

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

Checking $\mathbf{EX}\, f$ and $\mathbf{E}[f\, \mathbf{U}\, g]$ under fairness constraints is simpler. The set of all states which are the start of some fair computation is

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

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

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

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

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

8 Counterexamples and witnesses

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

In order to find the witness for some CTL formula we will need to examine the strongly connected components of the transition graph determined by the Kripke structure. We will say that two states s_1 and s_2 are *equivalent* if there is a path from s_1 to s_2 and also from s_2 to s_1. We will call the equivalence classes of this relation *strongly connected components*. We can form a new graph in which the nodes are the strongly connected components and there is an edge from one strongly connected component to another if and only if there is an edge from a state in one to a state in the other. It is easy to see that the new graph does not contain any proper cycles, i.e., each cycle in the graph is contained in one of the strongly connected components. Moreover, since we only consider finite Kripke structures, each infinite path must have a suffix that is entirely contained within a strongly connected component of the transition graph.

We start by considering the problem of how to find a witness for the formula $\mathbf{EG}\, f$ under the set of fairness constraints $H = \{h_1, \ldots, h_n\}$. We will identify each h_i with the set of states that make it true. Recall that the set of states that satisfy the formula $\mathbf{EG}\, f$ with the fairness constraints H is given by the formula

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

For brevity, we will use $\mathbf{EG}\, f$ to denote the set of states that satisfy $\mathbf{EG}\, f$ under the fairness constraints H. Given a state s in $\mathbf{EG}\, f$, we would like to exhibit a path π starting with s, which satisfies f in every state, and visits every set $h \in H$ infinitely often. In general, such a path will consist of a finite prefix followed by a repeating cycle. We construct the path incrementally by giving a sequence of prefixes of the path of increasing length until a cycle is found. At each step in the construction we must ensure that the current prefix can be extended to a fair path along which each state satisfies f.

This invariant is guaranteed by making sure that each time we add a state to the current prefix, the state satisfies $\mathbf{EG}\, f$.

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

Now, suppose we are given an initial state s satisfying $\mathbf{EG}\, f$. Then s belongs to the set of states computed in equation (1), so it must have a successor in $\mathbf{E}[f \,\mathbf{U}\, (\mathbf{EG}\, f) \wedge h]$ for each $h \in H$. In order to minimize the length of the witness path, we choose the first fairness constraint that can be reached from s. This is accomplished by testing the saved sets Q_i^h for increasing values of i until one is found that contains some successor t of s. Note that since $t \in Q_i^h$, it has a path to a state in $(\mathbf{EG}\, f) \wedge h$ and therefore t is in $\mathbf{EG}\, f$. If $i > 0$, we find a successor of t in Q_{i-1}^h. This is done by finding the set of successors of t, intersecting it with Q_{i-1}^h, and then choosing an arbitrary element of the resulting set. Continuing until $i = 0$, we obtain a path from the initial state s to some state in $(\mathbf{EG}\, f) \wedge h$. We then eliminate h from further consideration, and repeat the above procedure until all of the fairness constraints have been visited. Let s' be the final state of the path obtained thus far.

To complete a cycle, we need a non-trivial path from s' to the state t along which each state satisfies f. In other words, we need a witness for the formula $\{s'\} \wedge \mathbf{EX}\, \mathbf{E}[f \,\mathbf{U}\, \{t\}]$. If this formula is true, we have found the witness path for s. This case is illustrated in Figure 10. If the formula is false, there are several possible strategies. The simplest is to restart the procedure from the final state s'. Since $\{s'\} \wedge \mathbf{EX}\, \mathbf{E}[f \,\mathbf{U}\, \{t\}]$ is false, we know that s' is not in the strongly connected component of f containing t, however s' is in $\mathbf{EG}\, f$. Thus, if we continue this strategy, we must descend in the directed acyclic graph of strongly connected components, eventually either finding a cycle π, or reaching a terminal strongly connected component of f. In the latter case, we are guaranteed to find a cycle, since we cannot exit a terminal strongly connected component. This case is illustrated in Figure 11.

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

The witness procedure for $\mathbf{EG}\, f$ under fairness constraints H can be used to extend witnesses for $\mathbf{E}[f \,\mathbf{U}\, g]$ and $\mathbf{EX}\, f$ to infinite fair paths. Let $fair$ be the set of states that satisfy $\mathbf{EG}\, True$ under the fairness constraints H. We can compute $\mathbf{E}[f \,\mathbf{U}\, g]$ under H by using the standard CTL model checking algorithm (without fairness constraints) to compute $\mathbf{E}[f \,\mathbf{U}\, (g \wedge fair)]$. Similarly, We can compute $\mathbf{EX}\, f$ by using the standard CTL model checking algorithm to compute $\mathbf{EX}(f \wedge fair)$.

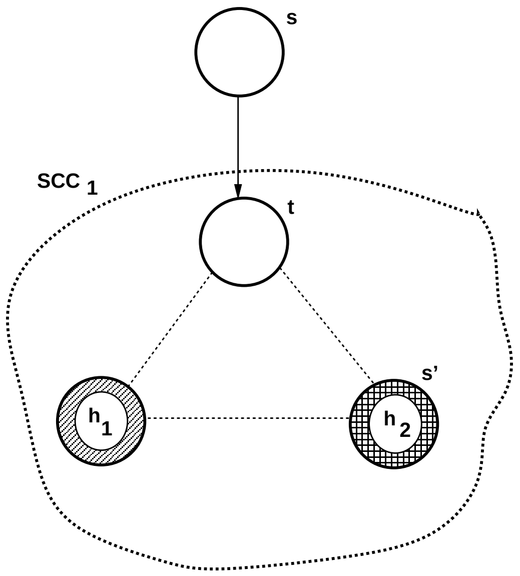


Fig. 10. Witness is in first strongly connected component

9 LTL Model Checking

In this section we consider the model checking problem for linear temporal logic. Let $\mathbf{A}\,f$ be a linear temporal logic formula. Thus, f is a *restricted path formula* in which the only state subformulas are atomic propositions. We wish to determine all of those states $s \in S$ such that $M, s \models \mathbf{A}\,f$. By definition $M, s \models \mathbf{A}\,f$ iff $M, s \models \neg\,\mathbf{E}\,\neg f$. Consequently, it is sufficient to be able to check the truth of formulas of the form $\mathbf{E}\,f$ where f is a restricted path formula. If the Kripke structure is represented explicitly as a state transition graph, this problem is known to be PSPACE-complete [74] in general.

Lichtenstein and Pnueli [56] developed an algorithm for the problem that was linear in the size of the model M and exponential in the length of the formula f. Although their algorithm was linear in the size of the model, it was still impractical for large examples because of the state explosion problem. As in the case of CTL model checking, representing the transition relation as an OBDD enables the procedure to be applied to much larger examples. The exponential complexity of their algorithm in terms of formula length is caused by a tableau construction which may require exponential space in the size of the formula. Fortunately, the tableau can also be represented by an OBDD. This leads to an additional reduction in space and time.

We begin with an informal description of the model checking algorithm. Given a formula $\mathbf{E}\,f$ and a Kripke structure M, we construct a special Kripke structure T called the *tableau* for the path formula f. This structure includes *every* path that satisfies f. By composing T with M, we find the set of paths that appear in both T and M. A state in M will satisfy $\mathbf{E}\,f$ if and only if it is the start of a path in the composition that satisfies f. The CTL model checking procedure described in Section 6 is used to find these states.

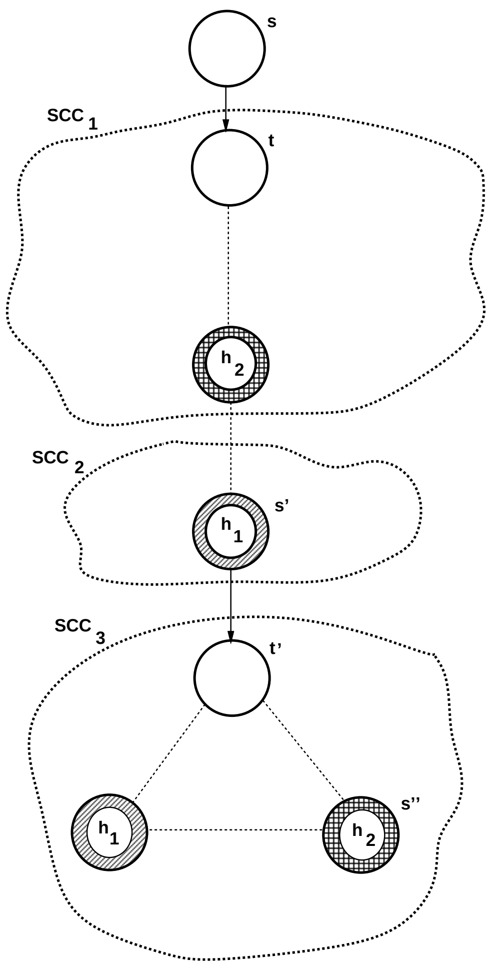


Fig. 11. Witness spans three strongly connected components

We now describe the construction of the tableau T in detail. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $T = (S_T, R_T, L_T)$ with AP_f as its set of atomic propositions. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively as follows:

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

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

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

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

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

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

Figure 12 gives the transition relation R_T for the formula $g = a\mathbf{U}b$. To reduce the number of edges, we connect two states σ and σ' with a bidirectional arrow if there is an edge from σ to σ' and also from σ' to σ. Each subset of $el(g)$ is a state of T. $sat(\mathbf{X}g) = \{1, 2, 3, 5\}$ since each of these states contains the formula $\mathbf{X}g$. $sat(g) = \{1, 2, 3, 4, 6\}$ since each of these states either contains b or contains a and $\mathbf{X}g$. There is a transition from each state in $sat(\mathbf{X}g)$ to each state in $sat(g)$ and from each state in the complement of $sat(\mathbf{X}g)$ to each state in the complement of $sat(g)$.

Unfortunately, the definition of R_T does not guarantee that *eventuality* properties are fulfilled. We can see this behavior in Figure 12. Although state 3 belongs to $sat(g)$, the path that loops forever in state 3 does not satisfy the formula g since b never holds on that path. Consequently, an additional condition is necessary in order to identify those paths along which f holds. A path π that starts from a state $\sigma \in sat(f)$ will satisfy f if and only if

- For every subformula $g\,\mathbf{U}\,h$ of f and for every state σ on π, if $\sigma \in sat(g\,\mathbf{U}\,h)$ then either $\sigma \in sat(h)$ or there is a later state τ on π such that $\tau \in sat(h)$.

In order to state the key property of the tableau construction, we must introduce some new notation. Let $\pi = s_0, s_1, \ldots$ be a path in a Kripke structure M, then $label(\pi) = L(s_0), L(s_1), \ldots$. Let $l = l_0, l_1, \ldots$ be a sequence of subsets of some set Σ and let $\Sigma' \subseteq \Sigma$. The *restriction* of l to Σ', denoted by $l\,|_{\Sigma'}$, is the sequence $l'_0, l'_1, \ldots$ where $l'_i = l_i \cap \Sigma'$ for every $i \geq 0$. The following theorem makes precise the intuitive claim that T includes every path which satisfies f.

Theorem 10. *Let T be the tableau for the path formula f. Then, for every Kripke structure M and every path π' of M, if $M, \pi' \models f$ then there is a path π in T that starts in a state in $sat(f)$, such that $label(\pi')\,|_{AP_f} = label(\pi)$.*

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

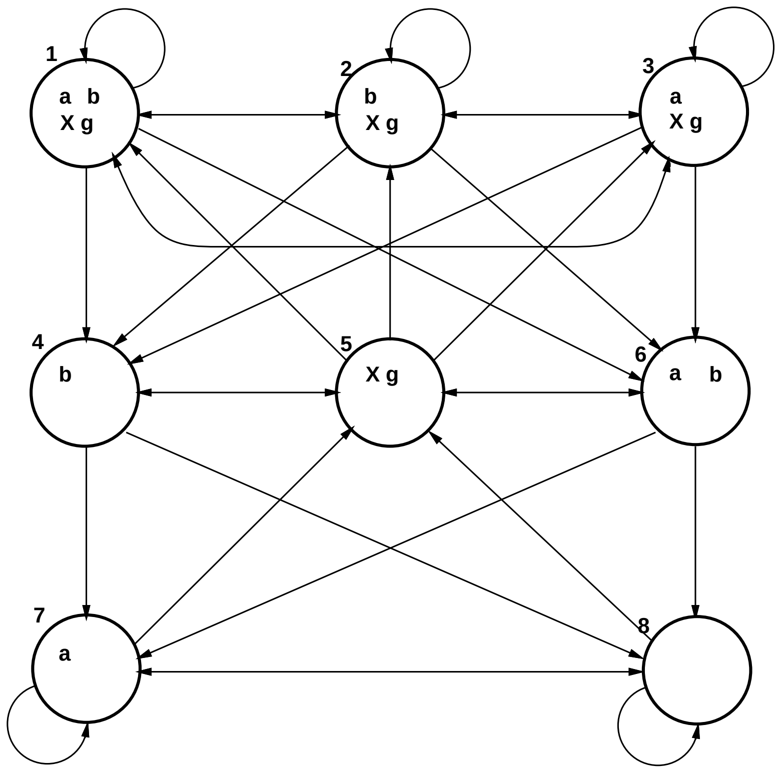


Fig. 12. Tableau for $a\,\mathbf{U}\,b$

- $S = \{(\sigma, \sigma') \mid \sigma \in S_T, \sigma' \in S_M \text{ and } L_M(\sigma') \cap AP_f = L_T(\sigma)\}$.
- $R((\sigma, \sigma'), (\tau, \tau'))$ iff $R_T(\sigma, \tau)$ and $R_M(\sigma', \tau')$.
- $L((\sigma, \sigma')) = L_T(\sigma)$.

P contains exactly the sequences π'' for which there are paths π in T and π' in M such that $label(\pi'') = label(\pi) = label(\pi') \mid_{AP_f}$. We extend the function sat to be defined over the set of states of the product P by $(\sigma, \sigma') \in sat(g)$ if and only if $\sigma \in sat(g)$.

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

$$\{sat(\neg(g\,\mathbf{U}\,h) \vee h) \mid g\,\mathbf{U}\,h \text{ occurs in } f\}. \tag{2}$$

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

Theorem 11. *$M, \sigma' \models \mathbf{E}\,f$ if and only if there is a state σ in T such that $(\sigma, \sigma') \in sat(f)$ and $P, (\sigma, \sigma') \models \mathbf{EG}\,True$ under fairness constraints $\{sat(\neg(g\,\mathbf{U}\,h) \vee h) \mid g\,\mathbf{U}\,h$ occurs in $f\}$.*

To illustrate this construction, we check the formula $g = a\,\mathbf{U}\,b$ on the Kripke structure M in Figure 13. The tableau T for this formula is given in Figure 12. If we compute the product P as described above, we obtain the Kripke structure shown in Figure 14. We use the CTL model checking algorithm to find the set V of states in $sat(g)$ that satisfy the formula $\mathbf{EG}\,true$ with the fairness constraint $sat(\neg(a\,\mathbf{U}\,b) \vee b)$. It is easy to see that the fairness constraint corresponds to the following set of states $\{(2,4'),(5,3'),(7,1'),(7,2')\}$. Thus, every state in Figure 14 satisfies $\mathbf{EG}\,true$. However, only $(2,4')$, $(3,1')$, and $(3,2')$ are in $sat(g)$, so the states $1'$, $2'$, and $4'$ of M satisfy $\mathbf{E}\,g = \mathbf{E}[a\,\mathbf{U}\,b]$.

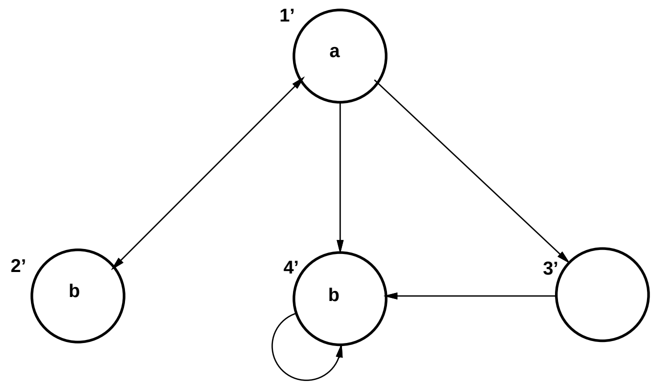


Fig. 13. Kripke Structure M

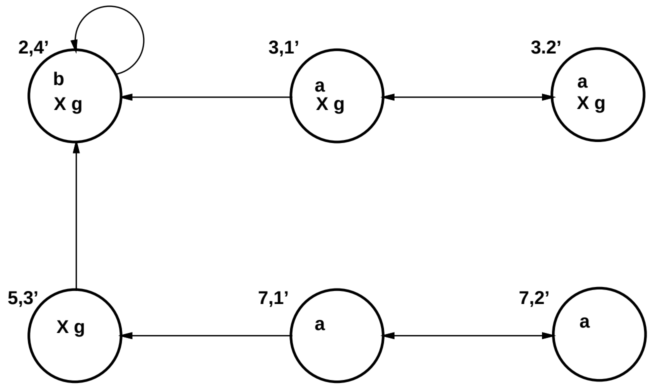


Fig. 14. The product P of the structure M and the tableau T

We now describe how the above procedure can be implemented using OBDDs. We assume that the transition relation for M is represented by an OBDD as in the previous section. In order to represent the transition relation for T in terms of OBDDs, we associate with each elementary formula g a state variable v_g. We describe the transition relation R_T as a boolean formula in terms of two copies $\bar{v}$ and $\bar{v}'$ of the state variables. The boolean formula is converted to an OBDD to obtain a concise representation of the tableau. When the composition P is constructed, it is convenient to separate out the state variables that appear in AP_f. The symbol $\bar{p}$ will be used to denote a boolean vector that assigns truth values to these state variables. Thus, each state in S_T will be represented by a pair $(\bar{p}, \bar{r})$, where $\bar{r}$ is a boolean vector that assigns values to the state variables that appear in the tableau but not in AP_f. A state in S_M will be denoted by a pair $(\bar{p}, \bar{q})$ where $\bar{q}$ is a boolean vector that assigns values to the state variables of M which are not mentioned in f. Thus, the transition relation R_P for the product of the two Kripke structures will be given by

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

We use the symbolic model checking algorithm that handles fairness constraints to find the set of states V that satisfy $\mathbf{EG}\, true$ with the fairness constraints given in (2). The states in V are represented by boolean vectors of the form $(\bar{p}, \bar{q}, \bar{r})$. Thus, a state $(\bar{p}, \bar{q})$ in M satisfies $\mathbf{E} f$ if and only if there exists $\bar{r}$ such that $(\bar{p}, \bar{q}, \bar{r}) \in V$ and $(\bar{p}, \bar{r}) \in sat(f)$.

10 Checking language containment

An alternative technique for verifying finite-state systems is based on showing language inclusion between finite ω-automata [48, 52, 77]. We model the system to be verified by an ω-automaton K_{sys}. The specification to be checked is given by a second ω-automaton K_{spec}. The system will satisfy its specification if the language accepted by K_{sys} is contained in the language accepted by K_{spec}, i.e. $\mathcal{L}(K_{sys}) \subseteq \mathcal{L}(K_{spec})$. In this section we show how symbolic model checking techniques can be used to decide language containment between ω-automata. Although there are many types of ω-automata, in this paper we only consider Büchi automata. Algorithms for other types of automata can be derived in a similar fashion from results in [25]. In general, checking language inclusion between two nondeterministic ω-automata is PSPACE-hard. For this reason we consider a restricted case of the general problem in which the specification automaton is deterministic. For simplicity we also require that both automata are complete.

A finite Büchi automaton is a 5-tuple $K = \langle S, s_0, \Sigma, \Delta, B \rangle$, where

- S is a finite set of *states*
- $s_0 \in S$ is the *initial state*
- Σ is a finite *alphabet*
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*
- $B \subseteq S$ is the *acceptance set.*

The automaton is *deterministic* if for all states $s, t_1, t_2 \in S$ and input symbols $\sigma \in \Sigma$, if $\langle s, \sigma, t_1 \rangle$ and $\langle s, \sigma, t_2 \rangle$ are two transitions in Δ, then $t_1 = t_2$. The automaton is *complete* if for every state $s \in S$ and for every symbol $\sigma \in \Sigma$, there is a state $s' \in S$ such that $(s, \sigma, s') \in \Delta$. An infinite sequence of states $s_0 s_1 s_2 \ldots \in S^\omega$ is a *path* of a Büchi automaton if there exists an infinite sequence $\sigma_0 \sigma_1 \sigma_2 \ldots \in \Sigma^\omega$ such that $\forall i \geq 0, (s_i, \sigma_i, s_{i+1}) \in \Delta$. A sequence $\sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ is *accepted* by a Büchi automaton if there is a corresponding path $s_0s_1s_2 \ldots \in S^\omega$ with the property that the set states which occur infinitely often in $s_0s_1s_2 \ldots$ contains at least one element of B. The set of sequences accepted by an automaton M is called the language of M and is denoted by $\mathcal{L}(M)$.

Let K and K' be two Büchi automata over the same alphabet Σ. Let $M(K, K')$ be a Kripke structure $(S \times S', R, L)$ over AP $= \{q, q'\}$, where q, q' are two new symbols and

$q \in L(\langle s, s' \rangle)$ *iff* $s \in B$.
$q' \in L(\langle s, s' \rangle)$ *iff* $s' \in B'$.
$\langle s, s' \rangle R \langle r, r' \rangle$ *iff* $\exists \sigma \in \Sigma : \langle s, \sigma, r \rangle \in \Delta$ and $\langle s', \sigma, r' \rangle \in \Delta'$.

Recall that in Section 6 we showed how to encode Kripke structures symbolically. In [25], it is shown that, if K' is deterministic,

$$\mathcal{L}(K) \subseteq \mathcal{L}(K') \Leftrightarrow M(K, K') \models \mathbf{A}(\mathbf{GF}q \Rightarrow \mathbf{GF}q')$$

Note that the formula above is not a CTL formula since there are temporal operators that are not immediately preceded by path quantifiers. However, it is equivalent to $\mathbf{AG}\,\mathbf{AF}\,q'$ ("infinitely often q'") under the fairness constraint "infinitely often q". Checking the above formula with the given fairness constraint can be handled using the techniques described in Section 6.

Theorem 12. *$\mathcal{L}(K) \subseteq \mathcal{L}(K')$ if and only if $M(K, K') \models \mathbf{AG}\,\mathbf{AF}\,q'$ with fairness constraint q.*

11 The SMV Model Checker

SMV ("*Symbolic Model Verifier*") [61] is a tool for checking that finite-state systems satisfy specifications given in CTL. It uses the OBDD-based symbolic model checking algorithm in Section 6. The language component of SMV is used to describe complex finite-state systems. Some of the most important features of the language are described below:

Modules: The user can decompose the description of a complex finite-state system into modules. Individual modules can be instantiated multiple times, and modules can reference variables declared in other modules. Standard visibility rules are used for naming variables in hierarchically structured designs. Modules can have parameters, which may be state components, expressions, or other modules. Modules can also contain fairness constraints which can be arbitrary CTL formulas (See Section 6).

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

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

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

We will not provide a formal syntax or semantics for the language here; these can be found in McMillan's thesis [61]. Instead, we consider a simple two process mutual exclusion program (figure 15). Each process can be in one of three code regions: the *non-critical region*, the *trying region*, and the *critical region*. Initially, both processes are in their non-critical regions. The goal of the program is to exclude the possibility that both processes are in their critical regions at the same time. We also require that a process, which wants to enter its critical region, will eventually be able to do so. A process indicates that it wants to enter its critical region by first entering its trying region. If one process is in its trying region and the other is in its non-critical region, the first process can immediately enter its critical region. If both processes are in their trying regions, the boolean variable *turn* is used to determine which process enters its critical region. If the value of *turn* is 0, then process 0 can enter its critical region and change the value of *turn* to 1. If the value of *turn* is 1, then process 1 can enter its critical region and change the value to 0. We assume that a process must eventually leave its critical region; however, it may remain in its non-critical region forever.

To describe the syntax of SMV in more detail, consider the program in Figure 15. Module definitions begin with the keyword `MODULE`. The module `main` is the top-level module. The module `prc` has formal parameters `state0`, `state1`, `turn`, and `turn0`. Variables are declared using the keyword `VAR`. In the example, `turn` is a boolean variable, while `s0` and `s1` are variables which can have one of the following values: `noncritical`, `trying` or `critical`. The `VAR` statement is also used to instantiate other modules as shown on lines 6 and 7. In our example, the module `prc` is instantiated twice, once with the name `pr0` and once with the name `pr1`. Since the keyword `process` is used in both cases, the global model is constructed by interleaving steps from `pr0` and `pr1`.

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

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

VAR

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

ASSIGN
init(turn) := 0;

FAIRNESS   !(s0 = critical)

FAIRNESS   !(s1 = critical)

SPEC   EF((s0 = critical) & (s1 = critical))

SPEC   AG((s0 = trying) -> AF (s0 = critical))

SPEC   AG((s1 = trying) -> AF (s1 = critical))

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

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

MODULE prc(state0, state1, turn, turn0)

ASSIGN

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

FAIRNESS   running
```

Fig. 15. SMV code for two process mutual exclusion program

Fairness constraints are given by `FAIRNESS` statements. Using the proposition `running` in the fairness constraint for module `prc` restricts the considered computations to only those in which each instance of `prc` is executed infinitely often. Without imposing additional constraints, the nondeterministic choice in line 27 would allow a process to remain in its critical region forever. The fairness constraints in lines 10 and 11 are used to prevent this possibility. The CTL properties to be verified are given as `SPEC` statements. The first specification (line 12) checks for a violation of the mutual exclusion requirement. The second and third specifications (lines 13 and 14) check that a process, which wants to enter its critical region, will eventually be able to do so. The last two specifications (lines 15 and 17) check whether processes must strictly alternate entry into their critical regions.

```
-- specification EF (s0 = critical & s1 = critical) is false

-- specification AG (s0 = trying -> AF s0 = critical) is true

-- specification AG (s1 = trying -> AF s1 = critical) is true

-- specification AG (s0 = critical -> A(s0 = critical U (... is false

-- specification AG (s1 = critical -> A(s1 = critical U (... is false

resources used:
user time: 1.15 s, system time: 0.3 s
BDD nodes allocated: 2405
BDD nodes representing transition relation: 56 + 1
```

Fig. 16. Output generated by SMV for mutual exclusion program

When SMV is run on the program in Figure 15 the output in Figure 16 is produced. Note that the mutual exclusion is not violated and that absence of starvation is guaranteed. Since the last two specifications are false, strict alternation of critical regions is unnecessary. SMV produced counterexample computation paths in the last two cases. One of the counterexamples is included in Figure 17. This counterexample demonstrates that process 0 can enter its critical region several times without process 1 entering its critical region. The computation path is described as a sequence of changes to state variable. Thus, if a state variable is not mentioned in a state it means that its value has not been changed. Although the first specification is false, no counterexample is generated. The negation of a formula with an existential path quantifier will have a universal path quantifier. Therefore, no single computation path can serve as a counterexample.

```
-- specification AG (s0 = critical -> A(s0 = critical U (... is false
-- as demonstrated by the following execution sequence
state 2.1: s0 = noncritical
           s1 = noncritical
           turn = 0

state 2.2: [executing process pr0]

state 2.3: [executing process pr0]
           s0 = trying

state 2.4: s0 = critical

state 2.5: [executing process pr0]

state 2.6: s0 = noncritical
           turn = 1

state 2.7: [executing process pr0]

state 2.8: [executing process pr0]
           s0 = trying

state 2.9: s0 = critical
```

Fig. 17. Counterexample for strict alternation of critical regions

12 A non-trivial example

This section briefly describes the formalization and verification of the cache coherence protocol described in the draft IEEE Futurebus+ standard (IEEE Standard 896.1–1991) [50]. We constructed a precise model of the protocol in the SMV language and then used model checking to show that the model satisfied a formal specification of cache coherence. In the process of formalizing and verifying the protocol, we discovered a number of errors and ambiguities. We believe that this is the first time that formal methods have been used to find nontrivial errors in a proposed IEEE standard. The result of our project is a concise, comprehensible and unambiguous model of the cache coherence protocol that should be useful to both the Futurebus+ Working Group members, who are responsible for the protocol, and to actual designers of Futurebus+ boards. Our experience demonstrates that hardware description languages and model checking techniques can be used to help design real industrial standards. For a more detailed treatment of this example, the reader is referred to the another paper [29] by the authors that deals exclusively with this topic.

Futurebus+ is a bus architecture for high-performance computers. The goal of the committee that developed Futurebus+ was to create a public standard for bus protocols that was unconstrained by the characteristics of any particular processor or device technology and that would be widely accepted and implemented by vendors. The cache coherence protocol used in Futurebus+ is required to insure consistency of data in hierarchical systems composed of many processors and caches interconnected by multiple bus segments. Such protocols are notoriously complex and, therefore, quite difficult to debug. Futurebus+ is, in fact, the first bus standard to include this capability. Although development of the cache coherence protocol began more than four years ago, to the best of our knowledge all previous attempts to validate the protocol have been based entirely on informal techniques [39]. In particular, no attempt has been made to specify the entire protocol formally or to analyze it using an automatic verification system.

The major part of the project involved developing a formal model for the cache coherence protocol in the SMV language and deriving CTL specifications for its correctness from the textual description of the protocol in the standard. Our model for the cache coherence protocol consists of 2300 lines of SMV code (not counting comments). The model is highly nondeterministic, both to reduce the complexity of verification (by hiding details) and to cover allowed design choices (indicated in the standard using the word *may*). By using SMV we were able to find several potential errors in the hierarchical protocol. The largest configuration that we verified had three bus segments, eight processors, and over 10^{30} states.

The Futurebus+ protocol, maintains coherence by having the individual caches *snoop*, or observe, all bus transactions. Coherence across buses is maintained using *bus bridges*. Special agents at the ends of the bridges represent remote caches and memories. In order to increase performance, the protocol uses *split transactions*. When a transaction is split, its completion is delayed and the bus is freed; at some later time, an explicit response is issued to complete the transaction. This facility makes it possible to service local requests while remote requests are being processed.

To demonstrate how the protocol works, we consider some example transactions for a single *cache line* in the two processor system shown in figure 18. A cache line is a series of consecutive memory locations that is treated as a unit for coherence purposes. Initially, neither processor has a copy of the line in its cache; they are said to be in the

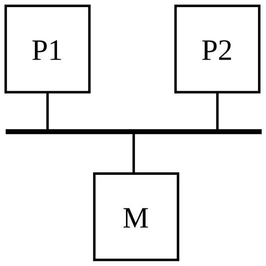


Fig. 18. Single bus system

invalid state. Processor P1 issues a *read-shared* transaction to obtain a readable copy of the data from memory M. P2 snoops this transaction, and may, if it wishes, also obtain a readable copy; this is called *snarfing*. If P2 snarfs, then at the end of the transaction, both caches contain a *shared-unmodified* copy of the data. Next, P1 decides to write to a location in the cache line. In order to maintain coherence, the copy held by P2 must be eliminated. P1 issues an *invalidate* transaction on the bus. When P2 snoops this transaction, it purges the line from its cache. At the end of the invalidate, P1 now has an *exclusive-modified* copy of the data. The standard specifies the possible states of the cache line within each processor and how this state is updated during each possible transaction.

We now consider a two-bus example to illustrate how the protocol works in hierarchical systems; see figure 19. Initially, both processor caches are in the invalid state.

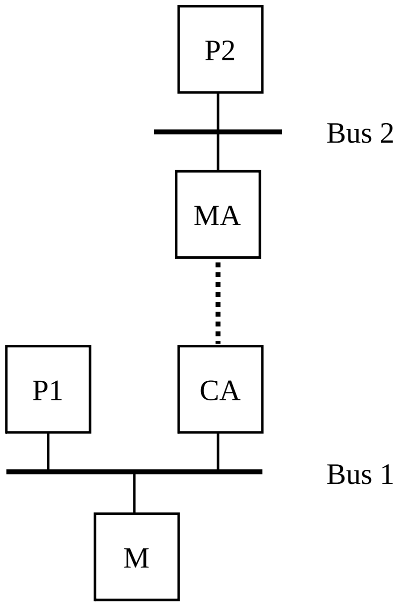


Fig. 19. Two bus system

If processor P2 issues a *read-modified* to obtain a writable copy of the data, then the memory agent MA on bus 2 splits the transaction, since it must get the data from the memory on bus 1. The command is passed down to the cache agent CA, and CA issues the read-modified on bus 1. Memory M supplies the data to CA, which in turn passes it to MA. MA now issues a *modified-response* transaction on bus 2 to complete the original split transaction. Suppose now that P1 issues a read-shared on bus 1. CA, knowing that a remote cache has an exclusive-modified copy, *intervenes* in the transaction to indicate that it will supply the data, and splits the transaction, since it must obtain the data from the remote cache. CA passes the read-shared to MA, which issues it on bus 2. P2 intervenes and supplies the data to MA, which passes it to CA. The cache agent performs a *shared-response* transaction which completes the original read-shared issued by P1. The standard contains an English description of the hierarchical protocol, but does not specify the interaction between the cache agents and memory agents.

The Protocol Specification [50] contains two sections dealing with the cache coherence protocol. The first, a description section, is written in English and contains an informal and readable overview of how the protocol operates, but it does not cover all scenarios. The second, a specification section, is intended to be the real standard. This section is written using *boolean attributes*. A boolean attribute is essentially a boolean variable together with some rules for setting and clearing it. The attributes are more precise, but they are difficult to read. The behavior of an individual cache or memory is given in terms of roughly 300 attributes, of which about 45 deal with cache coherence.

In order to make the verification feasible, we had to use a number of abstractions. First, we eliminated a number of the low level details dealing with how modules communicate. The most significant simplification was to use a model in which one step corresponds to one complete transaction on one of the buses in the system. This allowed us to hide all of the handshaking necessary to issue a command. Another example concerns the bus arbitration. The standard specifies two arbitration schemes, but we used a model in which a bus master is chosen completely nondeterministically. In addition, the standard describes how models behave in various exceptional situations, such as when a parity error is observed on the data bus. However, we did not consider such conditions.

The second class of simplifications was used to reduce the size of some parts of the system. For example, we only considered the transactions involving a single cache line. This is sufficient since transactions involving one cache line cannot affect the transactions involving a different cache line. Also, the data in each cache line was reduced to a single bit. The third class of simplifications involved eliminating the *read-invalid* and *write-invalid* commands. These commands are used in DMA transfers to and from memory. The protocol does not guarantee coherence for a cache line when a write-invalid transaction is issued for that line.

The last class of abstractions involved using nondeterminism to simplify the models of some of the components. For example, processors are assumed to issue read and write requests for a given cache line nondeterministically. Responses to split transactions are assumed to be issued after arbitrary delays. Finally, our model of a bus bridge is highly nondeterministic.

Figure 20 shows a part of the SMV program used to model the processor caches. This code determines how the state of the cache line is updated. Within this code, state components with upper-case names (`CMD`, `SR`, `TF`) denote bus signals visible to the cache, and components with lower-case names (`state`, `tf`) are under the control of the cache. The first part of the code (lines 3–13) specifies what may happen when an idle cycle occurs (`CMD=none`). If the cache has a shared-unmodified copy of the line, then the line may be nondeterministically kicked out of the cache unless there is an outstanding request to change the line to exclusive-modified. If a cache has an exclusive-unmodified copy of the line, it may kick the line out of the cache or change it to exclusive-modified.

The second part of the code (lines 15–26) indicates how the cache line state is updated when the cache issues a read-shared transaction (`master` and `CMD=read-shared`). This should only happen when the cache does not have a copy of the line. If the transaction is not split (`!SR`), then the data will be supplied to the cache. Either no other caches will snarf the data (`!TF`), in which case the cache obtains an exclusive-unmodified copy, or

```
 1  next(state) :=
 2    case
 3    CMD=none:
 4      case
 5      state=shared-unmodified:
 6        case
 7        requester=exclusive: shared-unmodified;
 8        1: {invalid, shared-unmodified};  -- Maybe kick line out of cache
 9        esac;
10      state=exclusive-unmodified: {invalid, shared-unmodified,
11        exclusive-unmodified, exclusive-modified};
12      1: state;
13      esac;
14    ...
15    master:
16      case
17      CMD=read-shared:  -- Cache issues a read-shared
18        case
19        state=invalid:
20          case
21          !SR & !TF: exclusive-unmodified;
22          !SR: shared-unmodified;
23          1: invalid;
24          esac;
25        ...
26        esac;
27      ...
28      esac;
29    ...
30    CMD=read-shared:  -- Cache observes a read-shared
31      case
32      state in {invalid, shared-unmodified}:
33        case
34        !tf: invalid;
35        !SR: shared-unmodified;
36        1: state;
37        esac;
38      ...
39      esac;
40    ...
41    esac;
```

Fig. 20. A portion of the processor cache model

some other cache snarfs the data, and everyone obtains shared-unmodified copies. If the transaction is split, the cache line remains in the invalid state.

The last piece of code (lines 30–39) tells how caches respond when they observe another cache issuing a read-shared transaction. If the observing cache is either invalid or has a shared-unmodified copy, then it may indicate that it does not want a copy of the line by deasserting its `tf` output. In this case, the line becomes invalid. Alternatively, the cache may assert `tf` and try to snarf the data. In this case, if the transaction is not split (`!SR`), the cache obtains a shared-unmodified copy. Otherwise, the cache stays in its current state.

Next, we discuss the specifications used in verifying the protocol. More exhaustive specifications are obviously possible; in particular, we have only tried to describe what cache coherence is, not how it is achieved. The first class of properties states that if a cache has an exclusive-modified copy of some cache line, then all other caches should not have copies of that line. The specification includes the formula

$$\mathbf{AG}(\mathit{p1.writable} \rightarrow \neg \mathit{p2.readable})$$

for each pair of caches *p1* and *p2*. Here, *p1.writable* is given in a `DEFINE` statement and is true when *p1* is in the exclusive-modified state. Similarly, *p2.readable* is true when *p2* is not in the invalid state.

Consistency is described by requiring that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}(\mathit{p1.readable} \wedge \mathit{p2.readable} \rightarrow \mathit{p1.data} = \mathit{p2.data})$$

Similarly, if memory has a copy of the line, then any cache that has a copy must agree with memory on the data.

$$\mathbf{AG}(\mathit{p.readable} \wedge \neg \mathit{m.memory\text{-}line\text{-}modified} \rightarrow \mathit{p.data} = \mathit{m.data})$$

The variable *m.memory-line-modified* is false when memory has an up-to-date copy of the cache line.

The final class of properties is used to check that it is always possible for a cache to get read or write access to the line.

$$\mathbf{AG}\,\mathbf{EF}\,\mathit{p.readable} \wedge \mathbf{AG}\,\mathbf{EF}\,\mathit{p.writable}$$

Finally, we describe two of the errors that we found while trying to verify the protocol. The first error occurs in the single bus protocol. Consider the system shown in figure 18. The following scenario is not excluded by the standard. Initially, both caches are invalid. Processor P1 obtains an exclusive-unmodified copy. Next, P2 issues a read-modified, which P1 splits for invalidation. The memory M supplies a copy of the cache line to P2, which transitions to the shared-unmodified state. At this point, P1, still having an exclusive-unmodified copy, transitions to exclusive-modified and writes the cache line. P1 and P2 are now inconsistent. This bug can be fixed by requiring that P1 transition to the shared-unmodified state when it splits the read-modified for invalidation. The change also fixes a number of related errors.

The second error occurs in the hierarchical configuration shown in Figure 21. P1, P2, and P3 all obtain shared-unmodified copies of the cache line. P1 issues an invalidate transaction that P2 and MA split. P3 issues an invalidate that CA splits. The bus bridge detects that an *invalidate-invalidate collision* has occurred. That is, P3 is trying

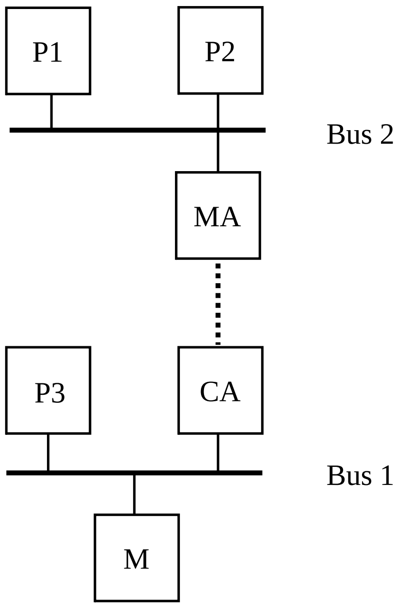


Fig. 21. Two bus system

to invalidate P1, while P1 is trying to invalidate P3. When this happens, the standard specifies that the collision should be resolved by having the memory agent invalidate P1. When the memory agent tries to issue an invalidate for this purpose, P2 sees that there is already a transaction in progress for this cache line and asserts a busy signal on the bus. MA observes this and acquires the *requester-waiting* attribute. When a module has this attribute, it will wait until it sees a completed response transaction before retrying its command. P2 now finishes invalidating and issues a modified-response. This is split by MA since P3 is still not invalid. However, MA still maintains the requester-waiting attribute. At this point, MA will not issue commands since it is waiting for a completed response, but no such response can occur. The deadlock can be avoided by having MA clear the requester-waiting attribute when it observes that P2 has finished invalidating.

13 Directions for Future Research

While symbolic representations have greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. Since model checkers must be used by engineers who are often not trained in formal methods, it is equally important to develop tool interfaces that are easy to use and that can provide information about the correctness of a circuit or protocol in a convenient format. In this section, we discuss several possible approaches that might be used in solving these problems.

13.1 Develop compositional reasoning techniques

The first approach exploits the *modular structure* of complex circuits. Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. If we can deduce that the system satisfies each local property, and if we know that the conjunction of the local properties implies the overall specification, then we can conclude that the complete system satisfies this specification as well. For instance, consider the problem of verifying a communications protocol that is modeled by three finite state processes: a transmitter, some type of network, and a receiver. Suppose that the specification for the system is that data is eventually transmitted correctly from the sender to the receiver. Such a specification might be decomposed into three local properties. First, the data should eventually be transferred correctly from the transmitter to the network. Second, the data should eventually be transferred correctly from one end of the network to the other. Finally, the data should eventually be transferred correctly from the network to the receiver. We might be able to verify the first of these local properties using only the transmitter and the network, the second using only the network, and the third using only the network and the receiver. By decomposing the verification in this way, we never have to compose all of the processes and therefore avoid the state explosion phenomenon.

There are a number of difficulties involved in developing a verifier that can support this style of reasoning. First, we must be able to check whether *every* system containing a given component satisfies a given local property. Since it is often the case that the local property is only true under certain conditions, we need to be able to make *assumptions* about the environment of the component when doing the verification. These assumptions, which represent requirements on other components, must also be checked in order to complete the verification. In addition, we must provide a method for checking that the conjunction of certain local properties implies a given specification. Several tools have been developed that permit this type of reasoning to be automated [47, 51, 57]. Currently, all of the research uses the temporal logic ACTL (CTL without existential path quantifiers). Hopefully, these methods can be extended to handle full CTL including existential path quantifiers. This is important in order to be able to verify the *existence* of executions that satisfy certain properties. For example, consider an network routing protocol. Such a protocol might not guarantee that any given message will be delivered, but it is a design error if the message cannot possibly be delivered. Research is also needed on automated and semi-automated methods for decomposing specifications into local properties.

13.2 Investigate the use of abstraction for reasoning about data paths

Verification techniques based on abstraction appear to be necessary for reasoning about concurrent systems that contain data paths. Traditionally, finite state verification methods have been used mainly for control-oriented systems. The symbolic methods make it possible to handle some systems that involve nontrivial data manipulation, but the complexity of verification is often high. This approach is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system. For example, in verifying the addition operation of a microprocessor, we might require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations *abstraction* can be used to reduce the complexity of model checking. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values. By extending the mapping to states and transitions, it is possible to produce an abstract version of the system under consideration. The abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level. It is possible to prove that any properties expressible in the logic ACTL that are satisfied by the abstract system must also be true of the actual system [30, 57].

In order to use this technique in practice, we must be able to construct the OBDD for an abstract model directly without first building the complete model. While representing an abstraction mapping with OBDDs is usually feasible, applying it to a specific model is often difficult. This problem can usually be avoided by starting with a high level description of the model (e.g. a program in a hardware description language) and combining the abstraction process with compilation. Frequently, by introducing additional OBDD variables, we can produce a single OBDD that represents an entire class of models. By performing the verification on this single representation, it is possible to check whether some property holds for the entire class. The complexity of verifying a class of models in this manner is often comparable to the complexity of verifying a single element in the class. This idea has been used to verify a pipelined arithmetic/logical unit with over 4000 state bits and 10^{1300} reachable states [30]. The verification times for this example scaled linearly with circuit size. Additional work is needed to extend these ideas and to determine how applicable they are to other types of systems.

13.3 Find ways of exploiting symmetry in circuits and protocols

Finite state concurrent systems frequently exhibit considerable symmetry. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols—anything that has a lot of replicated structure. It should be possible to use symmetry to reduce the size of the state space that must be explored by temporal logic model checking algorithms. Unfortunately, there has been relatively little research in the past on exploiting symmetry for this purpose. Most of the work on this problem has been performed by researchers investigating the reachability problem for Petri nets [64]. However, their work does not consider general temporal properties nor the complications that are caused by representing the state space using OBDDs. Recently, the use of symmetry in model checking has been investigated by several authors [32, 40].

We briefly outline the results on the use of symmetry obtained in [32]. Let G be a group of permutations acting on the state space S of the Kripke structure M. A permutation $\sigma \in G$ is said to be a *symmetry* of M if and only if it preserves the transition relation R. G is a *symmetry group* for the Kripke structure M if and only if every permutation $\sigma \in G$ is a *symmetry* for M. If s is an element of S, then the *orbit* of s is the set

$$\theta(s) = \{t \mid (\exists \sigma \in G)(\sigma s = t)\}$$

From each orbit $\theta(s)$ a representative (denoted by $rep(\theta(s))$) is selected.

If $M = (S, R, L)$ is a Kripke Structure and G is a symmetry group acting on M, it is possible to define a new structure $M_G = (S_G, R_G, L_G)$ called the *quotient model of M and G* in the following manner:

- The state set is $S_G = \{\theta(s) | s \in S\}$, the set of orbits of the states in S;
- The transition relation R_G has the property that $(\theta(s_1), \theta(s_2)) \in R_G$ if an only if $(s_1, s_2) \in R$;
- The labeling function L_G is given by $L_G(\theta(s)) = L(rep(\theta(s)))$.

An atomic proposition is *invariant* under the action of a symmetry group G, if the set of states labeled by the proposition is closed under the application of the permutations in G. It can be proved [32] that if h is a formula in the temporal logic CTL* and all of the atomic propositions in h are invariant under the symmetry group G, then h is true in M if and only if it is true in the quotient model M_G. Thus, it is possible to determine the correctness of properties in the original model M by checking them in the quotient model M_G. Since in the quotient model M_G there is only one representative from each orbit, the state space S_G will, in general, be much smaller than the the original state space S. In [32] a technique is described that permits M_G to be constructed without actually building M.

This approach is currently being testing on a simple cache coherency protocol based on the Futurebus+ IEEE standard. Previous research on verification of cache coherence protocols has made the simplifying assumption that there is only one cache line in the system [29, 62]. This assumption is necessary because the OBDDs that occur in verifying these protocols grow exponentially in the number of cache lines. By using symmetry, however, it is possible to avoid this assumption and reason about systems with multiple cache lines. Since different cache lines behave almost independently, the ordering of the cache lines is relatively unimportant and this results in a small quotient model. The initial results that have been obtained are encouraging. The size of the OBDDs that are needed to represent the model are, in some cases, reduced by an order of magnitude or more.

13.4 Develop techniques for verifying parameterized systems

A number of methods have been proposed for extending model checking based verification to parameterized designs that have an arbitrary number of similar or identical processes. 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. The first researchers to tackle this question were Browne, Clarke and Grumberg [28], 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 relationship 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-process 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 "by hand" in an *ad hoc* manner. Finite state methods cannot be used to check it because it is a map between states of a finite-state process and a process with an arbitrary number of states. A method without this disadvantage was proposed by Kurshan and McMillan [54], and independently by Wolper and Lovinfosse [80]. This method uses a process Q to act as an invariant, as the number of processes increases. If P represents one process in the system, then by showing that the possible executions of P composed with Q are *contained* in the possible executions of Q, it is possible to conclude by induction that Q adequately represents a system of any number of processes. Since both P composed with Q and Q are finite state processes, the containment relation (typically some form of language containment) can be checked automatically. This method is demonstrated in [61] by applying it to the Encore Gigamax cache consistency protocol. By slightly generalizing the model of one processor in the system, it is possible to obtain an invariant process to stand for any number of processors on a bus.

The induction technique has been generalized somewhat by Marelly and Grumberg [60] to apply systems generated by context-free grammar rules. The extra expressive power of these grammars may be of use for describing hierarchically structured systems, such as multi-level caches, or wide area networks. The main problem that requires additional research is that of constructing the invariant process, particularly finding automated techniques for this purpose. Currently, the invariant process must be constructed by hand. The counterexamples produced by the verifier are helpful for guiding the construction, but more powerful techniques would be useful.

13.5 Use partial orders to avoid the state explosion problem

Automatic verification techniques that explore the entire state space of a system are often hindered by an explosion of states resulting from the many possible permutations of concurrent events. Several researchers [46, 61, 69, 78] have investigated verification methods that avoid this explosion by disregarding the order of independent events. Petri nets are a natural model for this approach, since they make independence of events explicit, but other concurrency models can be used. The behavior of a Petri net can be characterized by an infinite unwinding of the net into an acyclic structure called an *occurrence net*. This structure is similar to the unwinding of a sequential program, but retains the concurrency which is inherent in the net. In [61] it is shown that the infinite unwinding can be effectively terminated when it is sufficient to represent all reachable states of the original net, even though there is no explicit representation of the states. Since the possible permutations of concurrent events are not enumerated, the occurrence net can be much smaller than the state graph of the system. This technique appears to be most promising for applications to asynchronous circuits and protocols. Preliminary experiments with a prototype verifier based on this approach have shown that the occurrence net method can be exponentially more efficient than state space enumeration.

13.6 Investigate techniques for verifying systems with real-time constraints

Many circuits and protocols have real-time constraints. Such systems are particularly difficult to verify because their correctness depends on the actual times at which events occur, in addition to the other properties that affect the correctness of systems without such constraints. There has been relatively little research on automatic verification techniques that are appropriate for this important class of finite state systems. Tools that are suitable for the verification of such systems are just beginning to be developed [3].

The real-time model that seems most useful in practice is based on discrete time and represents the passage of time with clock ticks. Burch [18] has investigated the relationship between this model of time and various continuous time models. He has proved that this model is a conservative approximation of the more realistic continuous time model. Since the complexity of automatic verification techniques based on continuous time models is much greater than for discrete time models, this should make it possible to handle much larger real-time systems.

13.7 Develop tool interfaces suitable for circuit and protocol designers

Since model checking avoids the construction of complicated proofs and provides a counterexample trace when some specification is not satisfied, circuit designers should find this technique much easier to learn and use than hardware verification techniques based on automated theorem proving or proof checkers. Several companies have already developed model checkers for languages that are widely used in industry like VHDL, Verilog, SDL, and Lotos.

One problem with the current system is how to make the specification language more expressive and easier to use. The ability to express time-bounded properties of synchronous circuits is important in many applications, and extensions to SMV have already been implemented to allow verification of such properties [23, 42] in the case of *discrete time*. It may be possible to develop better ways of representing circuits and protocols by using *continuous time* models. Such models provide a more precise specification of the timing characteristics of the circuit and, therefore, generate more accurate results. Model checking algorithms for continuous time are discussed in [2]. Unfortunately, current algorithms have high complexity and are not very useful in practice.

Finally, some type of *timing diagram* notation may be more natural for engineers than CTL. It may be possible either to translate timing diagrams systematically into temporal logic formulas or to check them directly using an algorithm similar to the one used by the model checker [72]. A similar problem arises in finding a good way to display the counterexamples that are generated when a formula is not true. This feature is invaluable for actually finding the source of a subtle error in a circuit design. However, current model checkers just print out a path in the state transition graph that shows how the error occurs. It is easy to imagine more perspicuous ways of displaying this information.

Acknowledgements

We would like to acknowledge Ken McMillan for his help on writing Section 8. We are also grateful to D. Beatty, K. Hamaguchi, and W. Marrero for their careful reading of this paper.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symp. on Logic in Computer Science*, pages 414–425, 1990.
3. R. Alur and T. A. Henzinger. Logics and models of real-time: A survey. In *Lecture Notes in Computer Science, Real-Time: Theory in Practice.* Springer-Verlag, 1992.
4. D. L. Beatty, R. E. Bryant, and C.-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference.* IEEE Computer Society Press, June 1991.
5. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
6. C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE International Conference on Computer Design*, 1990.
7. G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, C-31(3), March 1982.
8. S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In L. Claesen, editor, *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989.
9. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In DAC90 [36].
10. M. C. Browne and E. M. Clarke. Sml: A high level language for the design and verification of finite state machines. In *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France.* IFIP, September 1986.
11. M. C. Browne, E. M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proceedings of the 1985 International Conference on Computer Design*, Port Chester, New York, October 1985. IEEE.
12. M. C. Browne, E. M. Clarke, and D. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design.* Elsevier Science Publishers (North Holland), 1986.
13. 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.
14. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
15. R. E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.
16. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

17. R. E. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In Kurshan and Clarke [53].

18. J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie Mellon University, 1992.

19. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991. Winner of the Sidney Michaelson Best Paper Award.

20. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. To appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits.

21. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In DAC90 [36].

22. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

23. S. Campos. The priority inversion problem and real-time symbolic model checking. to appear, April 1993.

24. E. M. Clarke and I. A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354, pages 428–437. Springer-Verlag: Lecture Notes in Computer Science, 1988.

25. E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold and N. D. Jones, editors, *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1990.

26. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

27. 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.

28. E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing.*, pages 240–248. ACM, August 1986.

29. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.

30. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.

31. E. M. Clarke, S. Kimura, D. E. Long, S. Michaylov, S. A. Schwab, and J. P. Vidal. Symbolic computation algorithms on shared memory multiprocessors. In Suzuki [75].

32. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [35].

33. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Sifakis [73].

34. O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Kurshan and Clarke [53].

35. C. Courcoubetis, editor. *Proceedings of the Fifth Workshop on Computer-Aided Verification*, June/July 1993.

36. *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1990.

37. J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1989.

38. D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.

39. P. Dixon. Multilevel cache architectures. Minutes of the Futurebus+ Working Group meeting, December 1988.

40. E. Emerson and A. P. Sistla. Symmetry and model checking. In Courcoubetis [35].

41. E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.

42. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasen. Quantitative temporal reason. In Kurshan and Clarke [53].

43. E.A. Emerson and Chin Laung Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, January 1985.

44. M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the 1988 Proceedings of the IEEE International Conference on Computer Aided Design*. IEEE Computer Society Press, November 1988.

45. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

46. P. Godefroid. Using partial orders to improve automatic verification methods. In Kurshan and Clarke [53].

47. O. Grumberg and D. 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.

48. Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.–Feb. 1990.

49. G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic*. Methuen, London, 1977.

50. IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, March 1992. IEEE Standard 896.1–1991.

51. B. Josko. Verifying the correctness of AADL-modules using model checking. In de Bakker et al. [37].

52. R. P. Kurshan. Analysis of discrete event coordination. In de Bakker et al. [37].

53. R. P. Kurshan and E. M. Clarke, editors. *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

54. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1989.

55. L. Lamport. "Sometimes" is sometimes "Not Never". In *Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.

56. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985.

57. D. L. Long. *Model Checking, Abstraction, and Compositional Reasoning.* PhD thesis, Carnegie Mellon University, 1993.
58. 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.* Computer Science Press, 1981.
59. S. Malik, A. Wang, R. Brayton, and A Sangiovanni-Vincenteli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, pages 6–9, 1988.
60. R. Marelly and O. Grumberg. GORMEL—Grammar ORiented ModEL checker. Technical Report 697, The Technion, October 1991.
61. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.
62. K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In Suzuki [75].
63. B. Mishra and E.M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.
64. P.Huber, A. Jensen, L. Jepsen, and K. Jensen. Towards reachability trees for high-level petri nets. In G. Rozenberg, editor, *Advances on Petri Nets*, 1984.
65. C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. Clarke, editors, *Proc. CAV Workshop (also DIMACS Tech. Report 90-31)*, Rutgers University, NJ, June 1990.
66. C. Pixley, G. Beihl, and E. Pacas-Skewes. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of the International Conference on Computer Desgin*, pages 245–249, Cambridge, MA, October 1991.
67. C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*, pages 620–623, June 1992.
68. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
69. D. K. Probst and H. F. Li. Using partial order semantics to avoid the state explosion problem in asynchronous systems. In Kurshan and Clarke [53].
70. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
71. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Intl. Conf. on Computer Aided Design*, Santa Clara, Ca., November 1993.
72. R. Schlor and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *EDAC 93*, 1993.
73. J. Sifakis, editor. *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
74. A. P. Sistla and E.M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3):733–749, July 1986.
75. N. Suzuki, editor. *Shared Memory Multiprocessing.* MIT Press, 1992.
76. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.
77. H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.
78. A. Valmari. A stubborn attack on the state explosion problem. In Kurshan and Clarke [53].

79. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.
80. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In Sifakis [73].