Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking*

E. M. Clarke
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

O. Grumberg
Computer Science Dept.
The Technion
Haifa, 32000 Isreal

K. L. McMillan
Cadence Berkeley Labs.
1919 Addison Street, Ste. 303
Berkeley, CA 94704-1144, USA

X. Zhao
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

Model checking is an automatic technique for verifying sequential circuit designs and protocols. An efficient search procedure is used to determine whether or not the specification is satisfied. If it is not satisfied, our technique will produce a counterexample execution trace that shows the cause of the problem. We describe an efficient algorithm to produce counterexamples and witnesses for symbolic model checking algorithms. This algorithm is used in the SMV model checker and works quite well in practice. We also discuss how to extend our technique to more complicated specifications.

1 Introduction

Complex state-transition systems occur frequently in the design of sequential circuits and protocols. During the past ten years, researchers at Carnegie Mellon University have developed an alternative approach to verification called *temporal logic model checking* [4, 5]. 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.

One of the most important advantages of model checking over mechanical theorem provers or proof checkers for verification of circuits and protocols is its *counterexample facility.* Typically, the user provides a high level representation of the model and the specification to be checked. The model checking algorithm either terminates with the answer *true*, indicating that the model satisfies the specification, or gives a counterexample execution that shows why the formula is not satisfied. The counterexamples can be essential in finding subtle errors in complex designs.

*This research was sponsored in part by the National Science Foundation under Grant No. CCR-9217549, by the Semiconductor Research Corporation under Contract No. 94-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant No. F33615-93-1-1330.

Recently, the capability of model checking techniques has increased dramatically after the introduction of *ordered binary decision diagrams* (OBDDs) [1, 2, 12]. However, finding counterexamples is significantly more difficult when OBDDs are used in model checking instead of explicit state enumeration techniques, especially when fairness constraints are involved.

In this paper, we describe an efficient algorithm to produce counterexamples and witnesses for symbolic model checking algorithms. The algorithm is, in fact, the one that is used in the SMV model checker developed at Carnegie Mellon [12] and works quite well in practice. We show how the counterexample facility can be used to debug a subtle asynchronous circuit design. We also discuss how to extend our technique to more complicated temporal formulas.

A previous paper on debugging under the fairness constraints of the type considered in this paper appeared in [9]. They also gave a proof of the NP-completeness of this problem. However their algorithm is considerably different from ours and different trade-offs have been taken. In [9, 10], a set Fair+- is computed which is a better approximation to Fair than Fair+, the set used in the present paper. Then the algorithm finds the a state q in Fair+- which is nearest to the initial states and computes SCC(q). It checks to see if this is fair, i.e satisfies the fairness constraints;if so it builds a short cycle through q. If not, SCC(q) is eliminated from Fair+- and the computation is continued. Note that this algorithm guarantees a shortest path to a fair cycle is found.

On the other hand, our algorithm chooses some state s in Fair+ and then attempts to build a cycle satisfying all fairness constraints. This algorithm may actually leave the SCC of s, making the error trace unnecessarily long because a cycle through the first state s was missed. In general, in the method of Hojati/Brayton et al, more computation time is possible for finding a short error trace. They in fact guarantee that a shortest path to a fair cycle is found, but at the expense of computing strongly connected components(SCCs) for a number of states. The algorithm in the present paper is possibly faster since no SCC's are computed, but longer error traces may result. It remains to be determined if these speed-length trade-offs are actually seen in practice, since no comparisons have as yet been done.

2 The temporal logic CTL

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL or Computation Tree Logic [5]. Let P be the set of *atomic propositions,* then:

1. Every atomic proposition p in P is a formula in CTL.
2. If f and g are CTL formulas, then so are $\neg f$, $f \vee g$, $\mathbf{EX}\, f$, $\mathbf{E}[f\, \mathbf{U}\, g]$ and $\mathbf{EG}\, f$.

The semantics of a CTL formula is defined with respect to a *labeled state-transition graph.* A labeled state-transition graph is a 5-tuple $\mathbf{M} = (AP, S, L, N, S_0)$ where AP is a set of atomic propositions, S is a finite set of states, L is a function labeling each state with a set of atomic propositions, $N \subseteq S \times S$ is a transition relation, and S_0 is a set of initial states. A *computation path* is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $N(s_i, s_{i+1})$ is true for every i.

The propositional connectives $\neg$ and $\vee$ have their usual meanings of negation and disjunction. The other propositional operators can be defined in terms of these. $\mathbf{X}$ is the *nexttime* operator: $\mathbf{EX}\, f$ will be true in a state s of $\mathbf{M}$ if and only if s has a successor s' such that f is true at s'. $\mathbf{U}$ is the *until* operator: $\mathbf{E}[f\, \mathbf{U}\, g]$ will be true in a state s of $\mathbf{M}$ if and only if there exists a computation path starting in s and an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix. The operator $\mathbf{G}$ is used to express the *invariance* of some property over time: $\mathbf{EG}\, f$ will be true at a state s if there is a path starting at s such that f holds at each state on the path. If f is true in state s of structure $\mathbf{M}$, we write $\mathbf{M}, s \models f$. A CTL formula f is identified with the set $\{s | \mathbf{M}, s \models f\}$ of states that make f true. We use the following syntactic abbreviations for CTL formulas:

- $\mathbf{AX}\, f \equiv \neg\, \mathbf{EX}\, \neg f$ which means that f holds at all successor states of the current state.
- $\mathbf{EF}\, f \equiv \mathbf{E}[\text{true}\ \mathbf{U}\ \text{f}]$ which means that for some path, there exists a state on the path at which f holds.
- $\mathbf{AF}\, f \equiv \neg\, \mathbf{EG}\, \neg f$ which means that for every path, there exists a state on the path at which f holds.
- $\mathbf{AG}\, f \equiv \neg\, \mathbf{EF}\, \neg f$ which means that for every path, f holds in each state on the path.
- $\mathbf{A}[f\, \mathbf{U}\, g] \equiv \neg\, \mathbf{E}[\neg g\, \mathbf{U}\, \neg f \wedge \neg g] \wedge \neg\, \mathbf{EG}\, \neg g$ which means that for every path, there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix.

3 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 practice [4, 5]. 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.

Ordered binary decision diagrams (OBDDs) are a canonical form representation for boolean formulas [1]. 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.

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 is 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.

Our model checking algorithm is based on the standard fixpoint characterizations of the basic CTL operators. A *fixpoint* of $\tau : 2^S \to 2^S$ is a set $S' \subseteq S$ such that $\tau(S') = S'$. If τ is monotonic, it has a fixpoint S_0 that is a subset of every other fixpoint of τ. S_0 is called the *least fixpoint* of τ and is denoted by $\mathbf{lfp}\, f \left[\tau(f)\right]$. The *greatest fixpoint* of τ, $\mathbf{gfp}\, f \left[\tau(f)\right]$, can be defined similarly as the fixpoint of τ that is a superset of all other fixpoints. It can be shown that the least fixpoint $\mathbf{lfp}\, f \left[\tau(f)\right]$ is the limit of the sequence of approximations

$$\text{False}, \tau(\text{False}), \tau^2(\text{False}), \ldots, \tau^i(\text{False}), \ldots$$

and the greatest fixpoint $\mathbf{gfp}\, f \left[\tau(f)\right]$ is the limit of the sequence of approximations

$$\text{True}, \tau(\text{True}), \tau^2(\text{True}), \ldots, \tau^i(\text{True}), \ldots$$

When the state-transition graph is finite, both of these sequences are guaranteed to converge in a finite number of steps.

Each of the basic CTL operators can be characterized as a least or greatest fixpoint of some functional $\tau : 2^S \to 2^S$. In particular, it is shown in [4] that

- $\mathbf{E}[f\, \mathbf{U}\, g] = \mathbf{lfp}\, Z \left[g \vee \left(f \wedge \mathbf{EX}\, Z\right)\right]$, and
- $\mathbf{EG}\, f = \mathbf{gfp}\, Z \left[f \wedge \mathbf{EX}\, Z\right]$.

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:

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

Notice that these intermediate procedures take boolean formulas as their arguments, while *Check* takes a CTL formula as its argument. 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

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

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

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

In this case we can compute the sequence of approximations

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

for the least fixpoint as described above. 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 $Q_i = Q_{i+1}$, this process 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 $\mathbf{EG}$

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

If the OBDD for f is given, then the sequence of approximations for the greatest fixpoint can be used to compute the OBDD representation for the set of states that satisfy $\mathbf{EG}\, f$.

4 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

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

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 $\mathbf{E}[f\ \mathbf{U}\ (g \wedge \mathit{fair})]$ (without fairness constraints). Hence, we define

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

5 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**.

We start by considering the complexity of finding a good 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. Given a state s satisfying $\mathbf{EG}\, f$, we must exhibit a path π starting with s, such that f holds at each state, and every fairness constraint $h \in H$ is satisfied infinitely often along the path π. Since the witness is an infinite path, we must find a finite representation for it. It is easy to see that a witness can always be found that consists of a finite prefix followed by a repeating cycle. Each fairness constraint h_i is satisfied at least once on the cycle. Such a path is called a *finite witness*. The length of a finite witness is defined as the total length of the prefix and the cycle. It is desirable to find a finite witness with minimal length; however, this problem is NP-complete.

Theorem: If fairness constraints are permitted, finding a finite witness with minimal length for the formula $\mathbf{EG}\,$True is NP-complete.

Proof: It is relatively easy to see that this problem in NP. Finding a Hamiltonian cycle for a directed graph is known to be an NP-complete problem. Thus, it is sufficient to prove that the Hamiltonian cycle problem can be reduced to the minimal finite witness problem. Consider an instance of the Hamiltonian cycle problem for a directed graph with n nodes. This graph is treated as a state-transition graph and the set of fairness constraints $H = \{h_1, \ldots, h_n\}$ is selected so that each state satisfies a distinct fairness constraint. On any finite witness, each state must appear at least once on the cycle; hence, the length of the finite witness must be at least n. The length of the minimal finite witness is n if and only if the n states on the path form a Hamiltonian cycle. Thus, the Hamiltonian cycle problem reduces to finding a minimal finite witness and checking if this path has length n. This reduction can be performed in polynomial time. Consequently, the minimal finite witness problem is also NP-complete. □

Although we are unable to find the minimal finite witness easily, we still want to obtain a finite witness that is as short as possible. In order to accomplish this task, 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*(SCCs). We can form a new graph in which the nodes are the SCCs and there is an edge from one SCC 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 SCCs. Moreover, since we only consider finite Kripke structures, each infinite path must have a suffix that is entirely contained within a SCC of the transition graph.

Recall that the set of states that satisfy the formula $\mathbf{EG}\, f$ with the fairness constraints H is given by the formula

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

For brevity, we will use $\mathbf{EG}\, f$ to denote the set of states that satisfy $\mathbf{EG}\, f$ under the fairness constraints H. We construct the witness 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. 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 SCC 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 SCCs, eventually either finding a cycle π, or reaching a terminal SCC of f. In the latter case, we are guaranteed to find a cycle, since we cannot exit a terminal SCC.

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 SCCs

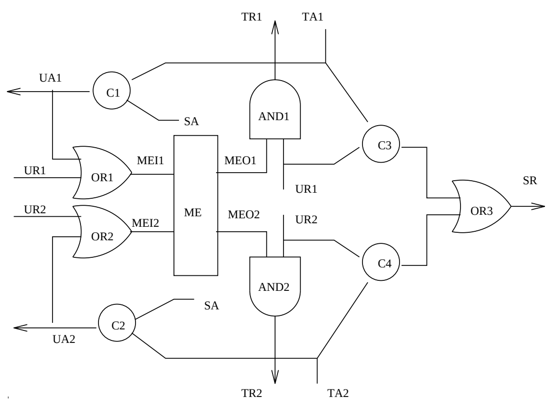

Figure 1: An asynchronous arbiter

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 $\mathbf{EX}(f \wedge fair)$.

In order to test the procedure for finding counterexamples when fairness constraints are used, we have examined an error in an arbiter design originally developed by Seitz [13]. The circuit is shown in Figure 1; it is designed to be *speed independent*, which means that each gate can take an arbitrarily long time to respond to its inputs. Fairness constraints are used to ensure that every gate eventually responds.

An attempt was made to verify the circuit using an explicit state model checker [6]. However, the attempt failed because the number of states was too large. In order to complete the verification, one of the input devices had to be disabled. By using symbolic model checking techniques, we are able to verify the original circuit without using any simplifying assumptions. The model contains 33,633 reachable states, and the entire verification takes only a few minutes.

We have verified several liveness properties which require that each request signal inevitably leads to an acknowledgement signal. Such properties can be easily represented by CTL formulas with the form $\mathbf{AG}(r \rightarrow \mathbf{AF}\, a)$, where r represents a *request* and a represents an *acknowledgment*. An error was discovered when the specification $\mathbf{AG}(tr1 \rightarrow \mathbf{AF}\, ta1)$ was checked. The algorithm given earlier in this section found a counterexample that was seventy eight states long and had a cycle with length thirty. The counterexample showed that the following execution sequence was possible. The circuit could reach a state where every node was low except *meo1* if the *ME* element took a long time to respond. When *ur1* was issued, *tr1, ta1, sr, sa* and *ua1* became true consecutively. Because of the long delay of the *OR1* gate, *mei1* remained low. Eventually, the *ME* element responded to its inputs and set *meo1* low. This caused *tr1* and *ta1* to become low. Next, *OR1* responded and *mei1* became high. Then, the *ME* element and the *AND1* gate caused *tr1* to become high again while *ta1* continued to be low. In this state, the formula $tr1 \rightarrow \mathbf{AF}\, ta1$ was false. Since *ua1* was already high, *ur1* could become low. This caused *tr1* to become low. The counterexample showed that *ur1* was always low. Therefore, *ta1* remained low as well. A correction for the error was proposed in [6], but will not be discussed here.

6 Counterexamples and Witnesses for CTL* Formulas

In the previous sections, we described how to perform model checking and find counterexamples or witnesses for CTL formulas. However, some temporal properties that are important for reasoning about sequential circuit designs and protocols cannot be expressed by CTL formulas. In these cases, an extension of CTL, called CTL*, is often used. 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). As before, let AP be the set of atomic propositions. 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. The logical connectives $\neg$ and $\vee$ have their usual meaning. The formula $\mathbf{E}(f)$ is true in a state when there exists a path from the state such that f holds along the path. Let $\pi = s_0, s_1, \ldots$ be a path. We use π^i to denote the *suffix* of π starting at s_i. A state formula holds along π when it is true in the first state s_0. $\mathbf{X}\, f$ holds along π when f holds along π^1. Finally, the formula $f\ \mathbf{U}\ g$ holds along π when there exists a $k \geq 0$ such that g holds on π^k and f holds along every π^j where $0 \leq j < k$. The following abbreviations are used in writing CTL* formulas:

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

In general, model checking is very expensive for CTL* formulas. However, for a large class of formulas which have the form $\mathbf{E} \bigvee_{i=1}^{n} \bigwedge_{j=1}^{n_i} (\mathbf{GF}\, p_{ij} \vee \mathbf{FG}\, q_{ij})$, efficient model checking algorithms exist [7]. Because

$$\mathbf{E} \bigvee_{i=1}^{n} \bigwedge_{j=1}^{n_i} (\mathbf{GF}\, p_{ij} \vee \mathbf{FG}\, q_{ij}) = \bigvee_{i=1}^{n} \mathbf{E} \bigwedge_{j=1}^{n_i} (\mathbf{GF}\, p_{ij} \vee \mathbf{FG}\, q_{ij}),$$

it is sufficient to check formulas having the form $\mathbf{E}\bigwedge_{j=1}^{n}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j)$. A fixed point characterization for these formulas is given in [7]

$$\begin{aligned}
&\mathbf{E}\bigwedge_{j=1}^{n}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \\
= \;&\mathbf{EF}\,\mathbf{gfp}\,Y\,\Big[\bigwedge_{j=0}^{n}((q_j \wedge \mathbf{EX}\,Y) \vee \mathbf{EX}\,\mathbf{E}[Y\,\mathbf{U}\,(p_j \wedge Y)])\Big].
\end{aligned}$$

By performing a computation that is similar to the one described in Section 4, we are able to check the restricted class of CTL* formulas mentioned above. The problem of finding witnesses for these formulas is more complicated. Suppose that we want find a witness for $s_0 \models \mathbf{E}\bigwedge_{j=1}^{n}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j)$. It is easy to see that

$$\begin{aligned}
&\mathbf{E}\bigwedge_{j=1}^{n}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \\
= \;&\mathbf{E}\bigwedge_{j=1}^{n-1}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \wedge \mathbf{GF}\,p_n \\
&\vee \mathbf{E}\bigwedge_{j=1}^{n-1}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \wedge \mathbf{FG}\,q_n.
\end{aligned}$$

Consequently, if $s_0 \models \mathbf{E}\bigwedge_{j=1}^{n-1}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \wedge \mathbf{FG}\,q_n$, it is sufficient to find a witness for this formula; otherwise, a witness must exist for $\mathbf{E}\bigwedge_{j=1}^{n-1}(\mathbf{GF}\,p_j \vee \mathbf{FG}\,q_j) \wedge \mathbf{GF}\,p_n$. If we continue this process for the remainder of the formula, we will eventually obtain a formula which has the form $\mathbf{E}\,\mathbf{FG}\,q_{i_1} \wedge \ldots \wedge \mathbf{FG}\,q_{i_k} \wedge \mathbf{GF}\,p_{j_1} \wedge \ldots \wedge \mathbf{GF}\,p_{j_{n-k}}$. Because

$$\begin{aligned}
&\mathbf{E}(\mathbf{FG}\,q_{i_1} \wedge \ldots \wedge \mathbf{FG}\,q_{i_k} \wedge \mathbf{GF}\,p_{j_1} \wedge \ldots \wedge \mathbf{GF}\,p_{j_{n-k}}) \\
= \;&\mathbf{EF}\,\mathbf{EG}(q_{i_1} \wedge \ldots \wedge q_{i_k} \wedge \mathbf{F}\,p_{j_1} \wedge \ldots \wedge \mathbf{F}\,p_{j_{n-k}}),
\end{aligned}$$

this formula is true if and only if the CTL formula $\mathbf{EG}(q_{i_1} \wedge \ldots \wedge q_{i_k})$ is true under the fairness constraints $p_{j_1}, \ldots, p_{j_{n-k}}$. A witness can be computed in exactly the same manner as in the last section.

7 Directions for Future Research

In this paper, we have described an efficient technique for generating counterexamples and witnesses for symbolic model checking algorithms. However, when the number of reachable states is very large, the counterexample can still be very long. Techniques for generating even shorter counterexamples will make symbolic model checking more useful in practice.

Finding a counterexample can sometimes take most of the execution time required for model checking. Additional research is needed to develop more efficient algorithms. This is particularly important because the model checking algorithm may need to be invoked several times in order to find the witness for a CTL* formula.

Another problem with the counterexample generated by the model checker is that it is sometimes hard to read. A more readable form will be helpful to engineers who are not familiar with model checking.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[8] 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.

[9] R. Hojati, R. K. Brayton, and R. P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. *Proceedings of the 5th international conference on computer aided verification*, June, 1993.

[10] R. Hojati, V. Singhal, and R. K. Brayton, Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment. *Memorandum No. UCB/ERL M94/12*, UC Berkeley, 1994.

[11] R. P. Kurshan. Analysis of discrete event coordination. In 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.

[12] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[13] C. L. Seitz. Ideas about arbiters. *Lambda*, 10(4), 1980.