RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE CONCURRENT SYSTEMS. (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE. R E BRYANT ET AL. DEC 87
UNCLASSIFIED CMU-CS-87-165 AFML-TR-87-1166
RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE CONCURRENT SYSTEMS

E.M. Clarke and O. Grumberg

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim

Approved for Public Release; Distribution is Unlimited
NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

CHAHIRA M. HOPPER
Project Engineer

RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER

EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.
Research on Automatic Verification of Finite-State Concurrent Systems

E. M. Clarke, O. Grumberg

Interim

FROM TO

1987 December

31

20

21

Unclassified/Unlimited

22a NAME OF RESPONSIBLE INDIVIDUAL

Chahira M. Hopper

(513) 255-7855

AFWAL/AAAT-3
Table of Contents

1. Introduction .................................................. 1
2. Computation Tree Logics .................................... 3
3. The CTL Model Checking Algorithm ....................... 5
4. Fairness Constraints .......................................... 7
5. An Example ...................................................... 9
6. Other Approaches ............................................. 13
7. Applications .................................................... 16
8. Conclusion ...................................................... 19
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5-1</td>
<td>Two process mutual exclusion program.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5-2</td>
<td>Transcript of model checker execution (without fairness constraint).</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5-3</td>
<td>Transcript of model checker execution (with fairness constraint).</td>
<td>12</td>
</tr>
<tr>
<td>Figure 7-1</td>
<td>Algorithm For Constructing Kripke Structure From Circuit</td>
<td>17</td>
</tr>
<tr>
<td>Figure 7-2</td>
<td>Kripke structure for unstable configuration of AND gate.</td>
<td>18</td>
</tr>
</tbody>
</table>
Research On Automatic Verification
Of Finite-State Concurrent Systems

E. M. Clarke and O. Grumberg
Carnegie Mellon University, Pittsburgh

1. Introduction

Temporal logics were first developed by philosophers for reasoning about the ordering of events in time without introducing time explicitly [Hughes & Creswell 77]. Although a number of different temporal logics have been studied, most have an operator like $G(f)$ 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 $G(\neg e_1 \lor \neg e_2)$. 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 [Emerson & Halpern 83] 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 [Hughes & Creswell 77].

Pnueli was apparently the first person to use temporal logic for specifying and verifying concurrent programs [Pnueli 77]. His approach involved proving desired properties of the program under consideration from a set of program axioms that described the behavior of the individual statements in the program. Proofs were usually constructed by hand, and this task was in general quite tedious. Since many concurrent programs can be viewed as communicating finite state machines, there was a strong possibility that at least some of these programs could be automatically verified. The first verification technique to exploit this observation was the $CTL$ model checking procedure developed by Clarke and Emerson in [Clarke & Emerson 81]. 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 [Gabbay et al 80] could be handled without changing the complexity of their algorithm. Handling fairness was an important step since the correctness of many concurrent

This research was partially supported by NSF Grant MCS-82-16706. The second author, O. Grumberg, is currently on leave from Technion, Haifa and is partially supported by a Weizmann postdoctoral fellowship.
algorithms depends critically on some assumption of this type: 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 [Quielle & Sifakis 81] 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 [Clarke et al. 86a] 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. Sistla and Clarke [Sistla & Clarke 86] 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 have shown how the temporal logic model checking procedure can be used for
verifying network protocols and sequential circuits ([Clarke et al. 86a], [Mishra & Clarke 85], [Browne et
al. 86], [Dill & Clarke 86], [Browne et al. 85], [Browne & Clarke 86], [Browne et al. 6b]). In the case of
sequential circuits two approaches have been developed for obtaining state transition graphs to analyze.
The first approach extracts a state graph directly from the circuit under an appropriate timing model of
circuit behavior. The second approach obtains a state transition graph by compilation from a high level
representation of the circuit in a Pascal-like programming language. In practice the model checking
procedure is able to check state transition graphs at a rate of 100 states per second for formulas of
reasonable length. It has been used successfully to find previously unknown errors in published designs
of asynchronous circuits.

Alternative approaches have been proposed by a number of other researchers. The approach used by
Kurshan [Kurshan 86] involves checking inclusion between two automata on infinite tapes. The first
machine represents the system that is being verified; the second represents its specification. Automata
on infinite tapes are used in order to handle fairness. Pnueli and Lichtenstein [Lichtenstein & Pnueli
85] 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 [Emerson & Lei 85] 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 have recently [Vardi & Wolper 86] shown how the model checking problem can be
formulated in terms of automata, thus relating the model checking approach to the work of Kurshan.
Although the model checking procedure discussed in this paper has already been used to discover some surprising errors in non-trivial programs, more work still remains to be done. Certainly the most serious problem is the state explosion problem. In analyzing a system of $N$ processes, the number of states in the global state graph may grow exponentially with $N$. Recent research indicates, however, that it may be possible to avoid this problem in some important cases. For instance, techniques developed in [Clarke et al 86b] may reduce the size of the state graph that needs to be searched when many of the processes are identical. It may also be possible to exploit the hierarchical structure of a complex concurrent program in order to reduce the number of states that need to be considered at any one level of abstraction [Mishra & Clarke 85].

This survey is organized as follows: Section 2 describes the syntax and semantics of the temporal logics that are used in this paper. In Section 3 we state the model checking problem and give an efficient algorithm for checking simple branching-time formulas. In Section 4 we discuss the issue of fairness and show how the algorithm of Section 3 can be extended to include fairness constraints. Section 5 demonstrates how the model checking algorithm can be used to debug a simple mutual exclusion program. In Section 6 we describe some alternative approaches for verifying systems of finite state concurrent processes. We analyze the complexity of checking linear temporal logic formulas and outline the techniques of Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] and Vardi and Wolper [Vardi & Wolper 86]. Additional applications to circuit and protocol verification are discussed in Section 7. The paper concludes in Section 8 with a discussion of some of the important remaining research problems like the state explosion problem.

2. Computation Tree Logics

In this paper finite state programs are modelled by labelled state-transition graphs, called Kripke structures [Hughes & Creswell 77]. 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. 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 the underlying computation tree. In 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 C11* ([Emerson & Clarke 81], [Emerson & Halpern 83], [Clarke et al 86a]) combines both branching-time and linear-time operators: a path quantifier, either A ("for all computation paths") or E ("for some computation path") 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. A state formula is either:

- $A$, if $A \in AP$.
- If $f$ and $g$ are state formulas, then $\neg f$ and $f \lor g$ are state formulas.
- If $f$ is a path formula, then $F(f)$ is a state formula.

A path formula is either:

- A state formula.
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $Xf$, and $f \lor Ug$ are path formulas.

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

$CTL$ ([Ben-Ari et al 83], [Clarke & Emerson 81]) is a restricted subset of $CTL^*$ that permits only branching-time operators—each path quantifier must be immediately followed by exactly one of the operators $G$, $F$, $X$, or $U$. More precisely, $CTL$ is the subset of $CTL^*$ that is obtained if the path formulas are restricted as follows:

- If $f$ and $g$ are state formulas, then $Xf$ and $Ug$ 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 $Af$ where $f$ is a path formula in which the only state subformulas that are permitted are atomic propositions. More formally, a path formula is either

- An atomic proposition
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $Xf$, and $f \lor Ug$ are path formulas.

We define the semantics of $CTL^*$ with respect to a structure $M = \langle S, R, L \rangle$, where

- $S$ is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total. We write $s_1 \to s_2$ to indicate that $(s_1, s_2) \in R$.
- $L: S \to \mathcal{P}(AP)$ 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.

We define a path in \( M \) to be a sequence of states, \( \pi = s_0, s_1, \ldots \) such that for every \( i \geq 0 \), \( s_i \rightarrow s_{i+1} \). \( \pi^i \) will denote the suffix of \( \pi \) starting at \( s_i \).

We use the standard notation to indicate that a state formula \( f \) holds in a structure: \( M,s \models f \) means that \( f \) holds at state \( s \) in structure \( M \). Similarly, if \( f \) is a path formula, \( M,\pi \models f \) means that \( f \) holds along path \( \pi \) in structure \( M \). 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. \( s \models A \) \iff \( A \in I(s) \).
2. \( s \models \neg f_1 \) \iff \( s \not\models f_1 \).
3. \( s \models f_1 \lor f_2 \) \iff \( s \models f_1 \) or \( s \models f_2 \).
4. \( s \models f_1(g_1) \) \iff \( \text{there exists a path } \pi \text{ starting with } s \text{ such that } \pi \models g_1 \).
5. \( \pi \models f_1 \) \iff \( s \) is the first state of \( \pi \) and \( s \models f_1 \).
6. \( \pi \models \neg g_1 \) \iff \( \pi \not\models g_1 \).
7. \( \pi \models g_1 \lor g_2 \) \iff \( \pi \models g_1 \) or \( \pi \models g_2 \).
8. \( \pi \models Xg_1 \) \iff \( \pi^1 \models g_1 \).
9. \( \pi \models g_1 \cup g_2 \) \iff \( \text{there exists a } k \geq 0 \text{ such that } \pi^k \models g_1 \text{ and for all } 0 \leq j < k, \pi^j \not\models g_1 \).

We will also use the following abbreviations in writing CTL* (CTL and LTL) formulas:

\( f \land g \equiv \neg (\neg f \lor \neg g) \)
\( Ff \equiv \text{true} \cup f \)
\( A(f) \equiv \neg E(\neg f) \)
\( Gf \equiv \neg F \neg f \)

In ([Lamport 80], [Emerson & Halpern 83]) it is shown 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 \( A(FGp) \). Likewise, there is no LTL formula that is equivalent to the CTL formula \( AG(\,EF\,p) \). The disjunction of these two formulas \( A(FGp) \lor AG(\,EF\,p) \) is a CTL* formula that is not expressible in either CTL or LTL.

3. The CTL Model Checking Algorithm

Let \( M = (S, R, I) \) be a finite Kripke structure. Assume that we want to determine which states in \( S \) satisfy the CTL formula \( f_\pi \). We will design our algorithm to operate in stages. The first stage processes all subformulas of \( f_\pi \) of length 1, the second stage processes all subformulas of length 2, and so on. At
the end of the $i^{th}$ stage, each state will be labeled with the set of all subformulas of length less than or equal to $i$ that are true in the state. We let the expression $\text{label}(s)$ denote this set for state $s$. When the algorithm terminates at the end of stage $n = \text{length}(f_n)$, we see that for all states and for all subformulas $f$ of $f_n$, $M.s \models f$ iff $f \in \text{label}(s)$.

Observe that $AX$ can be expressed in terms of $FX$ and that $AU$ can be expressed in terms of $FU$ and $FG$:

$$AXf_i \equiv \neg FX \neg f_i$$

$$A[f_i U f_j] \equiv \neg (F[-f_i U (\neg f_i \land \neg f_j)] \lor FG(\neg f_i)).$$

Thus, for the stage $i$ algorithm it is sufficient to be able to handle six cases, depending on whether $f$ is atomic or has one of the following forms: $\neg f_i$, $f_i \lor f_j$, $FXf_i$, $F[f_i U f_j]$ or $FGf_i$.

We will only consider the last two cases, since the others are straightforward.

To handle formulas of the form $f = F[f_i U f_j]$ we first find all of those states that are labeled with $f_i$. We then work backwards using the converse of the transition relation $R$ and find all of those states that can be reached by a path in which each state is labeled with $f_j$. All such states should be labeled with $f_i$. This step requires time $O(|S| + |R|)$.

The case in which $f = FGf_i$ is slightly more complicated and depends on the following observation.

**Lemma 1**: Let $M'$ be obtained from $M$ by deleting from $S$ all of those states at which $f_i$ does not hold and restricting $R$ and $I$ accordingly. Thus, $M' = (S', R', L')$ where $S' = \{ s \in S | M.s \models f_i \}$, $R' = R_{S'} \times S'$, and $L' = L_{S'}$. Then $M'.s \models FGf_i$ iff the following two conditions are satisfied:

1. $s \in S'$
2. there exists a path in $S'$ that leads from $s$ to some node $t$ in a non-trivial strongly connected component of the graph $(S', R')$.

**Proof**: Assume that $M'.s \models FGf_i$. Clearly $s \in S'$. Let $\pi$ be an infinite path starting at $s$ such that $f_i$ holds at each state on $\pi$. Since $M$ is finite, it must be possible to write $\pi$ as $\pi = \pi_1 \pi_2$ where $\pi_1$ is a finite initial segment and $\pi_2$ is an infinite suffix of $\pi$ with the property that each state on $\pi_2$ occurs
infinitely often. Obviously \( \pi_0 \) is contained in \( S' \). Let \( C \) be the set of states in \( \pi_1 \). \( C \) is a nontrivial strongly connected component of \( S' \). To see this, let \( s_1 \) and \( s_2 \) be states in \( C \). Pick some instance of \( s_1 \) on \( \pi_1 \). By the way in which \( \pi_1 \) was selected, we know that there is an instance of \( s_1 \) further along \( \pi_1 \). The segment from \( s_1 \) to \( s_2 \) lies entirely within \( C \) and hence within \( S' \). This segment is a finite path from \( s_1 \) to \( s_2 \) in \( S' \). Thus, both condition (1) and condition (2) are satisfied.

Next, assume that conditions (1) and (2) are satisfied. Let \( \pi_1 \) be the path from \( s \) to \( t \). Let \( \pi_2 \) be a finite path of length at least one that leads from \( t \) back to \( t \). The existence of \( \pi_2 \) is guaranteed since \( C \) is a non-trivial strongly connected component. All of the states on the infinite path \( \pi = \pi_1 \pi_2 \) satisfy \( f \). Since \( \pi \) is also a possible path starting at \( s \) in \( M \), we see that \( M, s \models EGf \). □

The algorithm for the case of \( f = EGf \) follows directly from the lemma. We construct the restricted Kripke structure \( M' = (S', R', L') \) as described in the statement of the lemma. We partition the graph \( (S', R') \) into strongly connected components and find those states that belong to nontrivial components. We then work backwards using the converse of \( R \) and find all of those states that can be reached by a path in which each state is labeled with \( f \). This step also requires time \( O(|S| + |R|) \).

In order to handle an arbitrary CTL formula \( f_0 \), we successively apply the state labeling algorithm to the subformulas of \( f_0 \), starting with the shortest, most deeply nested and work outward to include all of \( f_0 \). Since each pass takes time \( O(|S| + |R|) \) and since \( f_0 \) has \( \text{length}(f_0) \) different subformulas, the entire algorithm requires \( O(\text{length}(f_0) \cdot (|S| + |R|)) \).

Theorem 2: There is an algorithm for determining whether a CTL formula \( f_0 \) is true in state \( s \) of the structure \( M = (S, R, L) \) that runs in time \( O(\text{length}(f_0) \cdot (|S| + |R|)) \).

4. Fairness Constraints

In verifying concurrent systems, we are occasionally interested only in correctness along fair execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over an imperfect (or lossy) channel we may also wish to restrict the set of computation sequences; in this case the unfair execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver due to erratic behavior by the channel.

Roughly speaking, a fairness condition asserts that requests for service are granted "sufficiently often"
Different concepts of what constitutes a "request" and what "sufficiently often" should mean give rise to a variety of notions of fairness. Indeed, many different types of fairness and approaches to dealing with them have been proposed in the literature; we refer the reader to [Gabbay et al. 80], [Hannström 80], [Quielle & Sifakis 82], and [Lehmann et al. 81] for more extensive treatments. The text by Francez [Francez 86] also gives an excellent survey of the various types of fairness.

In this section we will show how to extend the CTL model checking algorithm to handle a simple but fundamental type of fairness in which certain predicates must hold infinitely often along every fair path. ([Clarke et al. 86a] shows how to handle a richer class of fairness constraints.) In this case it follows from [Emerson & Halpern 83] that correctness of fair executions cannot be expressed in CTL.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL^F, has the same syntax as CTL. But a structure is now a 4-tuple \( M = (S, R, L, F) \) where \( S, R, L \) have the same meaning as in the case of CTL, and \( F \) is a collection of predicates on \( S \), \( F \subseteq 2^S \). A path \( \pi \) is \( F \)-fair if the following condition holds: for each \( G \in F \), there are infinitely many states on \( \pi \) which satisfy predicate \( G \). CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths. The first step in checking CTL^F formulas is to determine the \( \text{fair strongly connected components} \) of the graph of \( M \). A strongly connected component is fair if it contains at least one state from each set in \( F \). Formally, let \( F = \{G_1, \ldots, G_n\} \) be a collection of subsets of \( S \). A strongly connected component \( C \) of the graph of \( M \) is fair iff for each \( G_i \) in \( F \), there is a state \( s_i \in (C \cap G_i) \).

**Lemma 3:** Given any finite structure \( M = (S, R, L, F) \) where \( F \) is a set of fairness constraints and a state \( s_0 \in S \), the following two conditions are equivalent:

1. There exists an \( F \)-fair path in \( M \) starting at \( s_0 \).
2. There exists a fair strongly connected component \( C \) of (the graph of) \( M \) such that there is a finite path from \( s_0 \) to a state \( t \in C \).

The proof is straightforward and is given in [Clarke et al. 86a]. We next extend our model checking algorithm to CTL^F. We introduce an additional proposition \( Q \), which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph associated with the structure and marking a component as \( \text{fair} \) if it contains at least one state from each \( G_i \) in \( F \). By the above lemma every state in a fair strongly connected component is the start of an infinite fair path. Thus, we label a state with \( Q \) iff there is a path from that state to some
node of a fair strongly connected component. As usual we design the algorithm so that after it terminates each state will be labeled with the subformulas of $f_0$ true in that state. We consider the two interesting cases where $f$ is a subformula of $f_0$ and either $f = F[f \cup f]$ or $f = FGf$. We assume that the states have already been labeled with the immediate subformulas of $f$ by an earlier stage of the algorithm.

1. $f = F[f \cup f]$: $f$ is true in a state iff the CTL formula $F[f \cup f \land Q]$ is true in that state, and this can be determined using the CTL model checker. Again, state $s$ is labeled with $f$ iff $f$ is true in that state.

2. $f = FG(f)$: To determine if $s \models FG(f)$ we use the procedure described in section 3 to check $s \models FG(f \land Q)$ in the structure with the additional proposition $Q$.

It is easy to see that the above algorithm runs in time $O(length(f_0) \cdot (|S| + |R|))$.

**Theorem 4:** There is an algorithm for determining whether a CTL$^F$ formula $f_0$ is true in state $s$ of the structure $M = (S, R, L, F)$ with $F$ as the set of fairness constraints that runs in time $O(length(f_0) \cdot (|S| + |R|))$.

5. **An Example**

In this section we illustrate how the model checker can be used to verify a simple, but not entirely trivial, concurrent program. The example is a two process mutual exclusion program that was manually proved correct using linear temporal logic by Owicki and Lamport in [Owicki & Lamport 82]. The program, expressed in a variant of the CSP programming language [Hoare 78], is shown in Figure 5-1. In this version of CSP processes may have global variables (e.g. $p1$ and $p2$), and assignments to such variables are assumed to be atomic. Since our verification technique can only be used to analyze finite state concurrent systems, we require that all variables be boolean and that all messages between processes be signals. Labels (e.g. $NC1$ and $NC2$) are used to indicate that flow of control has reached a particular point in some process. In our example there are two processes $S1$ and $S2$, and each process has three code regions: a *noncritical region* $NCi$ in which the process computes some data values that it wishes to share with the other process, a *trying region* $Ti$ in which the process executes a protocol to obtain entry into the critical section, and a *critical section* $CSi$ in which the process updates shared variables. To prevent a race condition that might result in unpredictable values being assigned to the shared variables, only one process is allowed to be in its critical section at any given time. Note that the two processes are different; hence this is not a *symmetric* solution to the mutual exclusion problem. When the CSP program is compiled a state graph with 77 states is obtained. Although this is not an extremely large state machine, it would nevertheless be quite tedious for a human to debug.
We initially run the verifier without any fairness constraints—See Figure 5-2. We first check to see if both processes are ever in their critical regions at the same time. This property is succinctly expressed by the CTL formula $\text{EF} (CS1 \land CS2)$. The verifier rapidly determines that the formula is false—hence, the program does not guarantee mutual exclusion. Time is measured in $1/60$ of a second. The first component measures user CPU time. The second component measures system CPU time. We next check for absence of deadlock. This is expressed by the formula $\text{AG} (\text{EF} (CS1 \lor CS2))$. The verifier determines that this formula is satisfied: thus, from any state that is reachable from the initial state it is always possible to get to either $CS1$ or $CS2$.

Absence of starvation for process 1 is expressed by the formula $\text{AG} (T1 \rightarrow \text{AF} (CS1))$. This property is not satisfied without a fairness constraint. The reason is quite simple. When we build the global state graph for the program we do not make any assumptions about the relative speeds of the two processes. Thus, the second process can make any number of steps between steps of the first process. In fact, the second process can even run forever, thereby preventing the first process from ever making another step. We can rule out the second type of behavior by means of fairness constraints which require that each process be given a chance to execute infinitely often. In Figure 5-3 we restart the verifier with several fairness constraints that prevent either process from remaining forever at the same statement while enabled to make a step. Under these assumptions the first process will never starve. However, the possibility of starvation still exists for the second process.

A good solution to the mutual exclusion problem should not require that processes alternate entry into their critical regions: $CS1, CS2, CS1, CS2, \ldots$. In order to test that the algorithm given in Figure 5-1 does not require strict alternation, we check the formula

$$\text{AG} (CS1 \rightarrow \text{A}[CS1 \cup (\neg CS1 \land A[\neg CS1 \cup CS2])]).$$

This formula asserts that if process 1 enters its critical section and subsequently leaves it, then it cannot enter it again until process 2 has entered its critical section. The verifier determines that the formula is false in less than a second. This example shows how the basic temporal operators, particularly the until operator, can be nested to express complicated timing properties.

Finally, the verifier has a counterexample feature (that is not shown in the transcripts). When this feature is enabled and the model checker determines that a formula is false, it will attempt to find a path in the state graph which demonstrates that the negation of the formula is true. For example, if the formula has the form $\text{AG} (\neg f)$, our system will produce a path to a state in which $\neg f$ holds. For instance, when the verifier determines that the last formula above is false, it prints out an execution of
s::[
  p1,p2: bool;
  NC1,NC2,T1,T2,T2a,CS1,CS2: label;
  S1,S2: process;
  S1::[
    p1:=false;
    *[true->
      <<NC1>> skip; --noncritical section 1
      p1:=true;
      <<T1>> *[p2->skip];
      <<CS1>> skip; --critical section 1
      p1:=false
    ]
  ]
]

S2::[
  p2:=false;
  *[true->
    <<NC2>> skip; --noncritical section 2
    p2:=true;
    <<T2>> *[p1->
      p2:=false;
      <<T2a>> *[p1->skip];
      p2:=true
    ];
    <<CS2>> skip; --critical section 2
    p2:=false
  ]
]

Figure 5-1: Two process mutual exclusion program.
CTL MODEL CHECKER (C version 2.5)

\[ |= \text{EF}(CS_1 \& CS_2). \]
The equation is FALSE.

time: (2 4)

\[ |= \text{AG}(\text{EF}(CS_1 | CS_2)). \]
The equation is TRUE.

time: (4 2)

\[ |= \text{AG}(T_1 \rightarrow \text{AF} CS_1). \]
The equation is FALSE.

time: (17 12)

Figure 5-2: Transcript of model checker execution (without fairness constraint).

Fairness constraint: \( \sim \text{NC}_1 \).
Fairness constraint: \( \sim \text{NC}_2 \).
Fairness constraint: \( \sim \text{CS}_1 \).
Fairness constraint: \( \sim \text{CS}_2 \).
Fairness constraint: \( \sim T_1 | P_2 \).
Fairness constraint: \( \sim |T_2 | p_1 \).
Fairness constraint: \( \sim T_2 | \sim |T_2 | p_2 \).
Fairness constraint: \( \sim T_2 | \sim |T_2 | p_1 \).
Fairness constraint: \( \sim T_2 | p_1 | T_2a \).
Fairness constraint: \( \sim T_2 | p_2 | T_2a \).

\[ |= \text{AG}(T_1 \rightarrow \text{AF} CS_1). \]
The equation is TRUE.

time: (10 0)

\[ |= \text{AG}(T_2 \rightarrow \text{AF} CS_2). \]
The equation is FALSE.

time: (29 9)

\[ |= \text{AG}(CS_1 \rightarrow \text{A}[CS_1 U (\sim CS_1 \& \text{A}[\sim CS_1 U CS_2])). \]
The equation is FALSE.

time: (38 17)

Figure 5-3: Transcript of model checker execution (with fairness constraint).
the mutual exclusion program in which process 1 enters its critical region, leaves, and reenters without process 2 entering its critical section in the meantime. This feature is quite useful for debugging purposes.

6. Other Approaches

Several papers have considered the model checking problem for linear temporal logic formulas. Let \( M = (S, \mathcal{R}, I) \) be a Kripke Structure with \( s_0 \in S \), and let \( \phi \) be a linear temporal logic formula. Thus, \( \phi \) is a restricted path formula in which the only state subformulas are atomic propositions. We wish to determine if \( M, s_0 \models \phi \). Notice that \( M, s_0 \models \phi \) iff \( M, s \models \neg E \neg \phi \). Consequently, it is sufficient to be able to check the truth of formulas of the form \( E \phi \) where \( \phi \) is a restricted path formula. In general, this problem is PSPACE-complete \([\text{Sistla \\ Clarke 86}]\). Although the proof of this PSPACE-completeness result is beyond the scope of our survey, it is easy to see that the model checking problem is NP-hard for formulas of the form \( E \phi \) where \( \phi \) is restricted path formula. We show that the directed Hamiltonian path problem is reducible to the problem of determining whether \( M, s \models \phi \) where

- \( M \) is a finite structure,
- \( s \) is a state in \( M \) and
- \( \phi \) is the assertion (using atomic propositions \( p_1, \ldots, p_n \)):

\[
E[p_1 \land \ldots \land p_n \land G(p_1 \rightarrow XG \neg p_1) \land \ldots \land G(p_n \rightarrow XG \neg p_n)].
\]

Consider an arbitrary directed graph \( G = (V, \mathcal{A}) \) where \( V = \{v_1, \ldots, v_n\} \). We obtain a structure from \( G \) by making proposition \( p_i \) hold at node \( v_i \) and false at all other nodes (for \( 1 \leq i \leq n \)), and by adding a source node \( u_1 \) from which all \( v_i \) are accessible (but not vice versa) and a sink node \( u_n \) which is accessible from all \( v_i \) (but not vice versa). Formally, let the structure \( M = (U, \mathcal{B}, I) \) consist of

\[
U = V \cup \{u_1, u_n\} \text{ where } u_1, u_n \in V;
\]

\[
B = \mathcal{A} \cup \{(u_i, v_i)|v_i \in V\} \cup \{(v_i, u_i)|v_i \in V\} \cup \{(u_i, u_n)\}; \text{ and}
\]

\( L \) is an assignment of propositions to states such that

- \( p_i \) is true in \( v_i \) for \( 1 \leq i \leq n \)
- \( p_i \) is false in \( v_i \) for \( 1 \leq i \leq n, i \neq j \)
- \( p_j \) is false in \( u_i, u_j \) for \( 1 \leq i \leq n \)
It is easy to see that $M, u \models f$ iff there is a directed infinite path in $M$ starting at $u$, which goes through all $v \in V$ exactly once and ends in the self loop though $u$. Note that the formula $f$ in the above construction has essentially the same size as the $G$ the graph $G$. Suppose that the length of the formula to be checked was known to be much smaller than the size of the Kripke structure under consideration. Would the complexity still be high in this case? A careful analysis by Lichtenstein and Pnueli [Lichtenstein & Pnueli 85] showed that although the complexity is apparently exponential in the length of the formula, it is linear in the size of the global state graph. We briefly describe their results below.

Let $f$ be a restricted path formula. The closure of $f$, $CL(f)$, is the smallest set of formulas containing $f$ and satisfying:

- $\neg f \in CL(f)$ iff $f \in CL(f)$
- If $f_1 \lor f_2 \in CL(f)$, then $f_1, f_2 \in CL(f)$
- If $Xf_1 \in CL(f)$, then $f_1 \in CL(f)$
- If $\neg Xf_1 \in CL(f)$, then $X\neg f_1 \in CL(f)$
- If $f_1 U f_2 \in CL(f)$, then $f_1, f_2, Xf_1[Uf_2] \in CL(f)$

It can be shown that the size of $CL(f)$ is $5 \cdot \text{length}(f)$.

An atom is a pair $A = (s_A, F_A)$ with $s_A \in S$ and $F_A \subseteq (L(f) \cup AP$ such that:

- for each proposition $Q \in AP$, $Q \in F_A$ iff $Q \in L(s_A)$
- for every $f_1 \in CL(f)$, $f_1 \in F_A$ iff $\neg f_1 \in F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 \lor f_2 \in F_A$ iff $f_1$ or $f_2 \in F_A$
- for every $\neg Xf_1 \in CL(f)$, $\neg Xf_1 \in F_A$ iff $X\neg f_1 \in F_A$
- for every $f_1, f_2 \in CL(f)$, $f_1 U f_2 \in F_A$ iff $f_1 \in F_A$ or $f_2, Xf_1[Uf_2] \in F_A$

Now, a graph $G$ is constructed with the set of atoms as the set of vertices. $(A, B)$ is an edge of $G$ iff $(s_A, s_B) \in R$ and for every formula $f_1$, if $Xf_1 \in F_A$, then $f_1 \in F_B$. An eventuality sequence is an infinite path $\pi$ in $G$ such that if $f_1 U f_2 \in F_A$ for some atom $A$ on $\pi$, then there exists an atom $B$, reachable from $A$ along $\pi$, such that $f_2 \in F_B$.

**Lemma 5:** $M, v \models Ff$ iff there exists an eventuality sequence starting at an atom $(x, F)$ such that $f \in F$.
A non-trivial strongly connected component $C$ of the graph $G$ is said to be self-fulfilling iff for every atom $A$ in $C$ and for every $f, f' \in F_A$ there exists an atom $B$ in $C$ such that $f \in F_B$.

**Lemma 6:** $M, s \models F f$ iff there exists an atom $A = (s, F)$ in $G$ such that $f \in F$ and there exists a path in $G$ from $A$ to a self-fulfilling strongly connected component.

Lemma 6 be used as the basis for a linear temporal logic model checking algorithm. This algorithm has the time complexity $O((|S| + |R|) \cdot 2^{\text{length}(f)})$. Lichtenstein and Pnueli further showed how this basic algorithm could be extended to handle a number of different notions of fairness with essentially the same complexity.

An alternative approach due to Vardi and Wolper [Vardi & Wolper 86] exploits the close relationship between linear temporal logic formulas and Büchi automata. A Büchi automata is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- $\Sigma$ is an alphabet.
- $S$ is a set of states.
- $\rho : S \times \Sigma \to 2^S$ is a non-deterministic transition function.
- $S_0 \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of designated states.

A run of $A$ on an infinite word $w = a_1 a_2 \ldots$ is a sequence $s_0 s_1 \ldots$ where $s_i \in S$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0 s_1 \ldots$ is accepting if there is some designated state that repeats infinitely often, i.e., for some $s' \in F$ there are infinitely many $i$'s such that $s_i = s$. The infinite word $w$ is accepted by $A$ if there is an accepting run of $A$ over $w$. The set of infinite words accepted by $A$ is denoted $L(A)$. The following theorem is proved in [Vardi & Wolper 86].

**Lemma 7:** For every linear temporal formula $\varphi$, a Büchi automata $A_\varphi$ can be constructed, where $\Sigma = 2^{4P}$ and $|S| \leq 2^{\text{length}(\varphi)}$, such that $L(A_\varphi)$ is exactly the set of computations satisfying the formula $\varphi$.

A Kripke Structure $M = (S, R, I)$ with initial state $s_0 \in S$ can be viewed as a Büchi automaton $A_M = (\Sigma, S, \{s_0\}, \rho, S)$ where $\Sigma = 2^{4P}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = I(s)$. Note that any infinite run of this automaton is accepting. $L(A_M)$ is the set of computations of $A_M$. Thus, in order to determine whether $M, s \models \varphi$ it is sufficient to check whether $L(A_M) \cap \Gamma(1, \varphi)$ is empty. This can be
determined by an automata theoretic construction with essentially the same time complexity as the Pnueli Lichtenstein algorithm.

One of the expected advantages of using linear temporal logic is that fairness constraints can be handled directly. However, if fairness constraints are included as part of the specifications, the formulas that must be checked will in general be quite large. For instance, consider a fairness constraint which requires that progress be made from any state in the program. The formula that expresses this property is

$$\forall \sigma \in S \left[ \neg \text{G}(\sigma(s)) \rightarrow \text{rest of specification} \right]$$

which has size $O(|S|)$. This problem was realized by Lichtenstein and Pnueli and by Vardi and Wolper. They in fact handle fairness by means of fairness constraints in a manner very similar to the way it is handled in [Clarke et al 86a]. Another problem with using linear temporal logic is that in general it is impossible to handle specifications which involve existential path quantifiers. Although it is possible to check simple formulas of the form $\exists \sigma \phi$ where $\phi$ is a restricted path formula, it is not possible to check formulas like $\forall \sigma (\exists \sigma \phi)$, which is used to express absence of deadlock in the example in section 5. Moreover, model checking for the full logic $\text{CTL}^*$ is no more difficult than for linear temporal logic as was shown by Emerson and Lei [Emerson & Lei 85].

**Theorem 8:** If we are given an algorithm $\text{AL}_{\text{LTL}}$ to solve the model checking problem for linear temporal logic, then we can construct an algorithm $\text{AL}_{\text{CTL}^*}$ for the full logic $\text{CTL}^*$ that has the same order of complexity as $\text{AL}_{\text{LTL}}$.

7. Applications

Sequential circuit verification is a natural application for the type of verifier discussed in this paper. Bochmann [Bochmann 82] was probably the first to realize the usefulness of temporal logic for describing the behavior of circuits. He verified an implementation of a self-timed arbiter using linear temporal logic and what he called "reachability analysis." The work of Malachi and Owicki [Malachi & Owicki 81] identified additional temporal operators required to express interesting properties of circuits and also gave specifications for a large class of modules used in self-timed circuits. Although these researchers contributed significantly toward developing an adequate notation for expressing the correctness of sequential circuits, the problem of mechanically verifying a circuit remained unsolved.

In [Mishra & Clarke 85] Clarke and Mishra showed how the EMC algorithm could be used to verify various temporal properties of asynchronous circuits. They developed a technique for extracting a state
graph directly from a wire-list description of the circuit (i.e., from a description of the circuit in terms of its components and their interconnections). The model checker was then used to show that state graph satisfied various specifications expressed in temporal logic. In this way they were able to determine that a self-timed queue element described in Seitz' chapter of Mead and Conway [Seitz 80] did not satisfy its specifications. Their work was later extended by Browne, Clarke, Dill, and Mishra [Browne et al 86] who showed, in general, how a mixed gate and switch level circuit simulator could be used to extract a state graph from a structural description of a sequential circuit. The basic simulation algorithm is shown in Figure 7-1. Circuits are usually designed under the assumption that certain input sequences and combinations will not occur. Their program exploits this observation to prevent a combinatorial explosion in the number of states that are generated, by allowing the user to specify a set of conditions under which the inputs can change.

(The procedure below uses a hash table that maps node value assignments to states. To construct the state machine, call this procedure on a node_value_assignment for the initial state.)

procedure BuildGraph(Node_value_assignment) return a state
begin
if there is a state for the node_value_assignment already in the table then return the state;
else
Create a new state;
Label state with nodes that have 1 values;
Store state and node values together in hash table;
for each possible input assignment do
Combine current values for internal nodes and input assignment into a new node_value_assignment;
Simulate one step to find a new node assignment;
Call BuildGraph recursively on new node assignment;
Add value returned by previous line to successors of current state;
end
end

Figure 7-1: Algorithm For Constructing Kripke Structure From Circuit

The circuit simulator in [Browne et al 86] used a unit-delay timing model in which the switching delays of all the transistors and gates are assumed to be equal. While a unit-delay model is satisfactory for synchronous circuits, it may not be appropriate for asynchronous circuits. In [Dill & Clarke 86] Dill and Clarke showed how Kripke structures could be extracted from a gate level description of a circuit under a model of circuit behavior that permitted arbitrary non-zero delays to be associated with the outputs of the gates. The basic idea behind their approach is quite simple. Consider an AND gate with
two inputs, $x$ and $y$, and a single output $z$. Assume that the gate is in an unstable configuration with $x$ low, $y$ high, and $z$ high. The Kripke structure for the circuit containing this gate will have a state corresponding to the unstable configuration as shown in Figure 7-2. The state will have a self-loop and a transition to another state representing a stable configuration in which the output is low. Fairness constraints, as described in Section 4, are used to ensure that the system doesn’t remain in an unstable configuration forever. In the case of the AND gate, it is sufficient to require that infinitely often $z = x \land y$.

\[ \text{Figure 7-2: Kripke structure for unstable configuration of AND gate.} \]

In practice, the arbitrary delay model is much too conservative. Many circuits are “almost speed independent”: They do not appear to be correct under a pure arbitrary delay model, but would work given reasonable assumptions about the relationships between the delays. When the circuit designer has a great deal of control over the magnitudes of circuit delays, exploiting more detailed knowledge of circuit timing can result in smaller and faster circuits. In fact, actual circuits often rely on such assumptions. In [Browne et al 85] and [Dill 86] a method is described for adding such assumptions to a circuit description and incorporating them into the state-graph construction. Possible timing constraints include constant upper and lower bounds on individual delays, and bounds on the differences between delays. Using constraints of this form, one can say for example: “the delay of the first AND gate is between 5 and 10 nanoseconds” or “the delay of the first AND gate is greater than the delay of the second AND gate.” The state graph constructed with respect to a particular set of delay assumptions rules out some circuit executions which would be allowed under an arbitrary delay model. Hence, formulas in C11, which might not have been true in an arbitrary delay model may be true with respect to particular delay assumptions (because all the counterexample paths are ruled out by the delay assumptions). This technique was applied to a patented asynchronous queue cell in [Browne et al 85]. The authors determined that the circuit did not meet its specifications under the arbitrary element delay
model. However, under the assumption that the input was slower than two of the circuit gates, they showed that the circuit met its temporal logic specifications.

An alternative approach obtains the state diagram by compilation from a specification of the original (synchronous) circuit in a simple programming language-like notation. Browne and Clarke ([Browne et al 86], [Browne & Clarke 86]) use a Pascal-like state machine description language called SML for this purpose. The language includes the standard control structures if, while, and loop/exit. A co-begin statement is also provided for simultaneous execution of statements in lock-step. Since SML programs will ultimately be implemented in hardware, the only data types permitted are boolean and (bounded) integer. The output of the SML compiler is a deterministic Moore Machine that can be automatically implemented as a PLA, PAL, or a ROM. The output can also be analyzed for correctness using the EMC algorithm. In [Browne 86] Browne describes a specialized version of the EMC algorithm that can check Moore machines much more rapidly than the original algorithm.

Another potential area of application is the verification of network communication protocols. The alternating bit protocol [Bartlet et al 69] for reliable transmission of messages by a noisy communication channel is a simple example of such an algorithm. By using the CTL model checking procedure it is possible to determine in a few seconds whether this protocol meets its specifications [Clarke et al 86a]. Sifakis at Grenoble [Quielle & Sifakis 81] and Kurshan at Bell Labs [Kurshan 86] have also considered applications involving network protocols. The delay assumptions mentioned above may be useful for describing the real-time behavior of such protocols.

8. Conclusion

Although the verification technique described in this paper has already been used to find some nontrivial errors in circuit designs and communications protocols, more research needs to be done before it will become a truly practical debugging tool for use by system designers. One problem is the expressibility of the underlying temporal logic. For circuit specification timing diagrams may be more natural to use than temporal logic formulas. Of course, temporal logic is more general since there is no analogue of negation, disjunction, or conjunction for timing diagrams. It may be possible to either systematically translate timing diagrams into temporal logic formulas or check them directly using an algorithm similar to the one used by the model checker. If so, this would simplify the task of specifying a complicated circuit and also allow the designer to be more confident that specifications actually mean what he thinks they mean.
The most important problem, however, is the state explosion problem. There are several different strategies for handling this problem. In verifying asynchronous circuits, for example, buggy circuits sometimes result in much larger state graphs than correct circuits. This happens because the activity in the circuit is much more disordered after an error has occurred. One possible solution in this case is to run the program which builds the state graph and the model checker as co-routines, creating states only as they need to be referenced by the model-checker. In [Dill 86] this technique is called lazy state generation, by analogy to lazy evaluation in programming language implementations. By using this method, an error could be discovered and reported after constructing only a small part of the entire state graph; this would not only speed up the verification process, it would also make it possible to verify some circuits which could not be verified if the entire graph had to be constructed.

Another approach to the state explosion problem is to exploit the hierarchical structure of complex finite state concurrent systems. If an appropriate subset of C11 is used ([Mishra & Clarke 85], [Clarke et al. 86b]), then lower level subcircuits can be simplified by "hiding" some of their internal nodes (more precisely, making it illegal to use them in temporal logic formulas) and merging groups of states that become indistinguishable into single state. Preliminary research in [Mishra & Clarke 85] indicates that by using this technique it may be possible to cut-down dramatically on the number of states that need to be examined.

Finally, special techniques may be appropriate for concurrent systems that are composed of many identical processes. Consider, for example, a distributed mutual exclusion algorithm for processes arranged in a ring network in which mutual exclusion is guaranteed by means of a token that is passed around the ring ([Dijkstra 85], [Kurshan 85], [Martin 85]). A strategy that is often used for debugging such systems is to consider first a reduced system with one or two processes. If it is possible to show that the reduced system is correct and if the individual processes are really identical, then one is tempted to conclude that the entire system will be correct. In [Clarke et al. 86b] an attempt is made to provide a solid theoretical basis that will prevent fallacious conclusions in arguments of this type. The authors describe a temporal logic called Indexed CTL*, or ICTL* for specifying networks of identical processes. The logic includes all of CTL* with the exception of the nexttime operator; in addition, it permits formulas of the form $\land_i f_i(t)$ and $\lor_i f_i(t)$ where $f_i(t)$ is a formula in which all of the atomic propositions are subscripted by $i$. A Kripke structure for a family of $N$ identical processes may be obtained as a product of the state graphs of the individual processes. Instances of the same atomic proposition in different processes are distinguished by using the number of the process as a subscript; thus, $\phi_i$ represents the instance of atomic proposition $\phi$ associated with process $i$. 
Since a closed formula of the new logic cannot contain any atomic propositions with constant index values, it is impossible to refer to a specific process by writing such a formula. Hence, changing the number of processes in a family of identical processes should not affect the truth of a formula in the logic. This intuitive idea is made precise by introducing a new notion of bisimulation [Milner 79] between two Kripke structures with the same set of indexed propositions but different sets of index values. It is possible to prove that if two structures correspond in this manner, a closed formula of Indexed CTL* will be true in the initial state of one if and only if it is true in the initial state of the other.

These ideas are illustrated in [Clarke et al 86b] by considering the distributed mutual exclusion algorithm mentioned above. The atomic proposition $c_i$ is true when the $i$-th process is in its critical region, and the atomic proposition $d_i$ is true when the $i$-th process is delayed waiting to enter its critical region. A typical requirement for such a system is that a process waiting to enter its critical region will eventually do so. This condition is easily expressed in ICTL* by the formula $\bigwedge AG(d_i \Rightarrow A F c_i)$. The results of [Clarke et al 86b] can be used to show that exactly the same ICTL* formulas hold in a network with 1000 processes as hold in a network with two processes. The EMC algorithm can be used to check automatically that the above formula holds in networks of size two and conclude that it will also hold in networks of size 1000. At present this methodology has only been partially automated, however. The bisimulation must be established by hand and this generally requires some representation of the larger Kripke structure. Several researchers are attempting to find a way of automating this phase in a manner that avoids building the larger Kripke structure.

Other techniques for avoiding the state explosion problem are being investigated by Kurshan and Wolper. In Kurshan's system [Kurshan 83] this problem is handled by using a homomorphism to collapse a large state machine into a much smaller one while preserving those properties that are important for verification. Since Kurshan does not use temporal logic formulas for specification, he has no analogue of the indexed formulas or of the bisimulation theorem used in [Clarke et al 86b]. Wolper [Wolper 86] considers a logic somewhat like ICTL* for reasoning about programs that are data-independent; however, his indexed variables range over data elements, not over processes. Also, there is no notion of correspondence between structures in his work. Some ultimate limitations on this type of reasoning are discussed in Apt and Kozen [Apt & Kozen 86].
References


END
FILMED
MARCH, 1988
DTIC
RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE
CONCURRENT SYSTEMS. (U) CARNEGIE-MELLON UNIV PITTSBURGH
PA DEPT OF COMPUTER SCIENCE. R E BRYANT ET AL. DEC 87
UNCLASSIFIED CMU-CS-87-105 AFWAL-TR-87-1166
<table>
<thead>
<tr>
<th>DTIC ACCESSION NUMBER</th>
<th>AD-A188 618</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCUMENT IDENTIFICATION</td>
<td>AFWAL-TR-87-1166</td>
</tr>
<tr>
<td>DATE ACCESSIONED</td>
<td>FEB 09 1988</td>
</tr>
<tr>
<td>DATE RETURNED</td>
<td>8 2 05 102</td>
</tr>
<tr>
<td>DATE RECEIVED IN DTIC</td>
<td>DEC 1987</td>
</tr>
<tr>
<td>DISTRIBUTION STATEMENT</td>
<td>DTIC ELECTED</td>
</tr>
<tr>
<td>DISTRIBUTION STAMP</td>
<td>A-1</td>
</tr>
</tbody>
</table>

This document has been approved for public release and sale.

DTIC TAB

UNANNOTCED

JUSTIFICATION

NTIS GRAI

DTIC TAB

UNANNOUNCED

JUSTIFICATION

BY

DISTRIBUTION / AVAILABILITY CODES

DIST

AVAIL AND/OR SPECIAL

A-1

QUALITY MANUFACTURED

PREVIOUS EDITION MAY BE USED UNIL STOCK IS EXHAUSTED.

DOCUMENT PROCESSING SHEET

DTIC FORM 70A

DECEMBER 83 STOIC IS (V

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

PREVIOUS EDITION MAY BE USED UNTIL STOCK IS EXHAusted.

DOCUMENT PROCESSING SHEET

DTIC FORM 70A

DECEMBER 83 STOIC IS (V

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

PREVIOUS EDITION MAY BE USED UNTIL STOCK IS EXHAusted.

DOCUMENT PROCESSING SHEET

DTIC FORM 70A

DECEMBER 83 STOIC IS (V

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

PREVIOUS EDITION MAY BE USED UNTIL STOCK IS EXHAusted.
RESEARCH ON AUTOMATIC VERIFICATION OF FINITE-STATE CONCURRENT SYSTEMS

E.M. Clarke and O. Grumberg
Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987
NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

CHAHIRA M. HOPPER
Project Engineer

RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER

EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.
**Title**: Research on Automatic Verification of Finite-State Concurrent Systems

**Authors**: E. M. Clarke, O. Grumberg

**Type of Report**: Interim

**Date of Report**: 1987 December

**Page Count**: 31

**Source of Funding Numbers**:
- Program: 61101E
- Project: 4976
- Task: 00
- Work Unit: 01

**Subject Terms**: (Continue on reverse if necessary and identify by block number)

**Abstract**: (Continue on reverse if necessary and identify by block number)

**Distribution/Availability of Report**: Approved for public release; distribution is unlimited.

**Monitoring Organization**: Air Force Wright Aeronautical Laboratories

**Procurement Instrument Identification Number**: F33615-84-K-1520

**Name of Responsible Individual**: Chahira M. Hopper

**Telephone**: (513) 255-7865

**Office Symbol**: AFWAL/AAAT-3
Table of Contents

1. Introduction .................................................. 1
2. Computation Tree Logics ............................. 3
3. The CTL Model Checking Algorithm .............. 5
4. Fairness Constraints .................................. 7
5. An Example .................................................. 9
6. Other Approaches .......................................... 13
7. Applications .................................................... 16
8. Conclusion ..................................................... 19
List of Figures

Figure 5-1: Two process mutual exclusion program. 11
Figure 5-2: Transcript of model checker execution (without fairness constraint). 12
Figure 5-3: Transcript of model checker execution (with fairness constraint). 12
Figure 7-1: Algorithm For Constructing Kripke Structure From Circuit 17
Figure 7-2: Kripke structure for unstable configuration of AND gate. 18
Research On Automatic Verification Of Finite-State Concurrent Systems

E. M. Clarke and O. Grumberg
Carnegie Mellon University, Pittsburgh

1. Introduction

Temporal logics were first developed by philosophers for reasoning about the ordering of events in time without introducing time explicitly [Hughes & Creswell 77]. Although a number of different temporal logics have been studied, most have an operator like $G(f)$ 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 $G(\neg e_1 \land \neg e_2)$. 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 [Emerson & Halpern 83] 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 [Hughes & Creswell 77].

Pnueli was apparently the first person to use temporal logic for specifying and verifying concurrent programs [Pnueli 77]. His approach involved proving desired properties of the program under consideration from a set of program axioms that described the behavior of the individual statements in the program. Proofs were usually constructed by hand, and this task was in general quite tedious. Since many concurrent programs can be viewed as communicating finite state machines, there was a strong possibility that at least some of these programs could be automatically verified. The first verification technique to exploit this observation was the CTL model checking procedure developed by Clarke and Emerson in [Clarke & Emerson 81]. 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 [Gabbay et al. 80] could be handled without changing the complexity of their algorithm. Handling fairness was an important step since the correctness of many concurrent

---

This research was partially supported by NSF Grant MCS-82-16706. The second author, O. Grumberg, is currently on leave from Technion, Haifa, and is partially supported by a Weizmann postdoctoral fellowship.
algorithms depends critically on some assumption of this type: 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 [Quielle & Sifakis 81] 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 Sisla [Clarke et al 86a] 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. Sisla and Clarke [Sisla & Clarke 86] 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 have shown how the temporal logic model checking procedure can be used for verifying network protocols and sequential circuits ([Clarke et al 86a], [Mishra & Clarke 85], [Browne et al 86], [Dill & Clarke 86], [Browne et al 85], [Browne & Clarke 86], [Browne et al 8b]). In the case of sequential circuits two approaches have been developed for obtaining state transition graphs to analyze. The first approach extracts a state graph directly from the circuit under an appropriate timing model of circuit behavior. The second approach obtains a state transition graph by compilation from a high level representation of the circuit in a Pascal-like programming language. In practice the model checking procedure is able to check state transition graphs at a rate of 100 states per second for formulas of reasonable length. It has been used successfully to find previously unknown errors in published designs of asynchronous circuits.

Alternative approaches have been proposed by a number of other researchers. The approach used by Kurshan [Kurshan 86] involves checking inclusion between two automata on infinite tapes. The first machine represents the system that is being verified; the second represents its specification. Automata on infinite tapes are used in order to handle fairness. Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] 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 [Emerson & Lei 85] 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 have recently [Vardi & Wolper 86] shown how the model checking problem can be formulated in terms of automata, thus relating the model checking approach to the work of Kurshan.
Although the model checking procedure discussed in this paper has already been used to discover some surprising errors in non-trivial programs, more work still remains to be done. Certainly the most serious problem is the state explosion problem. In analyzing a system of $N$ processes, the number of states in the global state graph may grow exponentially with $N$. Recent research indicates, however, that it may be possible to avoid this problem in some important cases. For instance, techniques developed in [Clarke et al 86b] may reduce the size of the state graph that needs to be searched when many of the processes are identical. It may also be possible to exploit the hierarchical structure of a complex concurrent program in order to reduce the number of states that need to be considered at any one level of abstraction [Mishra & Clarke 85].

This survey is organized as follows: Section 2 describes the syntax and semantics of the temporal logics that are used in this paper. In Section 3 we state the model checking problem and give an efficient algorithm for checking simple branching-time formulas. In Section 4 we discuss the issue of fairness and show how the algorithm of Section 3 can be extended to include fairness constraints. Section 5 demonstrates how the model checking algorithm can be used to debug a simple mutual exclusion program. In Section 6 we describe some alternative approaches for verifying systems of finite state concurrent processes. We analyze the complexity of checking linear temporal logic formulas and outline the techniques of Pnueli and Lichtenstein [Lichtenstein & Pnueli 85] and Vardi and Wolper [Vardi & Wolper 86]. Additional applications to circuit and protocol verification are discussed in Section 7. The paper concludes in Section 8 with a discussion of some of the important remaining research problems like the state explosion problem.

2. Computation Tree Logics

In this paper finite state programs are modelled by labelled state-transition graphs, called Kripke structures [Hughes & Creswell 77]. 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. 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 the underlying computation tree. In 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 $CTL^*$ ([Emerson & Clarke 81], [Emerson & Halpern 83], [Clarke et al 86a]) combines both branching-time and linear-time operators; a path quantifier, either $A$ ("for all computation paths") or $E$ ("for some computation path") 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 $\text{CTL}^*$: 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. A state formula is either:

- $A$, if $A \in AP$.
- If $f$ and $g$ are state formulas, then $\neg f$ and $f \lor g$ are state formulas.
- If $f$ is a path formula, then $\forall(f)$ is a state formula.

A path formula is either:

- A state formula.
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $Xf$, and $fUg$ are path formulas.

$\text{CTL}^*$ is the set of state formulas generated by the above rules.

$\text{CTL}$ ([Ben-Ari et al 83], [Clarke & Emerson 81]) is a restricted subset of $\text{CTL}^*$ that permits only branching-time operators—each path quantifier must be immediately followed by exactly one of the operators $G$, $F$, $X$, or $U$. More precisely, $\text{CTL}$ is the subset of $\text{CTL}^*$ that is obtained if the path formulas are restricted as follows:

- If $f$ and $g$ are state formulas, then $Xf$ and $fUg$ 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 $A f$ where $f$ is a path formula in which the only state subformulas that are permitted are atomic propositions. More formally, a path formula is either

- An atomic proposition
- If $f$ and $g$ are path formulas, then $\neg f$, $f \lor g$, $Xf$, and $fUg$ are path formulas.

We define the semantics of $\text{CTL}^*$ with respect to a structure $M = \langle S, R, L \rangle$, where

- $S$ is a set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total. We write $s_1 \rightarrow s_2$ to indicate that $(s_1, s_2) \in R$.
- $L: S \rightarrow 2^{AP}$ 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.

We define a path in \( M \) to be a sequence of states, \( \pi = s_0 s_1 \ldots \) such that for every \( i \geq 0 \), \( s_i \rightarrow s_{i+1} \). \( \pi^i \) will denote the suffix of \( \pi \) starting at \( s_i \).

We use the standard notation to indicate that a state formula \( f \) holds in a structure: \( M, s \models f \) means that \( f \) holds at state \( s \) in structure \( M \). Similarly, if \( f \) is a path formula, \( M, \pi \models f \) means that \( f \) holds along path \( \pi \) in structure \( M \). 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. \( s \models A \Rightarrow A \in I(s) \)
2. \( s \models \neg f_1 \Rightarrow s \not\models f_1 \)
3. \( s \models f_1 \lor f_2 \Rightarrow s \models f_1 \) or \( s \models f_2 \)
4. \( s \models E(g_1) \Rightarrow \) there exists a path \( \pi \) starting with \( s \) such that \( \pi \models g_1 \)
5. \( \pi \models f_1 \Rightarrow s \) is the first state of \( \pi \) and \( s \models f_1 \)
6. \( \pi \models \neg g_1 \Rightarrow \pi \not\models g_1 \)
7. \( \pi \models g_1 \lor g_2 \Rightarrow \pi \models g_1 \) or \( \pi \models g_2 \)
8. \( \pi \models Xg_1 \Rightarrow \pi^1 \models g_1 \)
9. \( \pi \models g_1 \cup g_2 \Rightarrow \) there exists a \( k \geq 0 \) such that \( \pi^k \models g_1 \) and for all \( 0 \leq j < k \), \( \pi^j \models g_1 \).

We will also use the following abbreviations in writing \( \text{CTL}^* \) (\( \text{CTL} \) and \( \text{LTL} \)) formulas:

- \( f \land g \equiv \neg(\neg f \lor \neg g) \)
- \( Ff \equiv \text{true} \cup f \)
- \( A(f) \equiv \neg E(\neg f) \)
- \( Gf \equiv \neg F\neg f \).

In ([Lamport 80], [Emerson & Halpern 83]) it is shown that the three logics discussed in this section have different expressive powers. For example, there is no \( \text{CTL} \) formula that is equivalent to the \( \text{LTL} \) formula \( M(FGp) \). Likewise, there is no \( \text{LTL} \) formula that is equivalent to the \( \text{CTL} \) formula \( AG(EPp) \). The disjunction of these two formulas \( M(FGp) \lor AG(EPp) \) is a \( \text{CTL}^* \) formula that is not expressible in either \( \text{CTL} \) or \( \text{LTL} \).

3. The CTL Model Checking Algorithm

Let \( M = (S, R, I) \) be a finite Kripke structure. Assume that we want to determine which states in \( S \) satisfy the \( \text{CTL} \) formula \( f_r \). We will design our algorithm to operate in stages. The first stage processes all subformulas of \( f_r \) of length 1, the second stage processes all subformulas of length 2, and so on. At
the end of the $i^{th}$ stage, each state will be labeled with the set of all subformulas of length less than or equal to $i$ that are true in the state. We let the expression $\text{label}(s)$ denote this set for state $s$. When the algorithm terminates at the end of stage $n = \text{length}(\phi)$, we see that for all states and for all subformulas $f$ of $\phi$, $M.s \models f$ iff $f \in \text{label}(s)$.

Observe that $\AX f_i \equiv \neg \EX \neg f_i$

$\AU[f_i \lor f] \equiv \neg (F[\neg f_i \lor (\neg f_i \land \neg f_i)] \lor \EF(\neg f_i))$.

Thus, for the stage $i$ algorithm it is sufficient to be able to handle six cases, depending on whether $f$ is atomic or has one of the following forms: $\neg f_i, f_i \lor f_i, \EX f_i, \EF[f_i \lor f], \EF f_i$ or $\EF f_i$.

We will only consider the last two cases, since the others are straightforward.

To handle formulas of the form $f = \EF[f_i \lor f]$ we first find all of those states that are labeled with $f_i$. We then work backwards using the converse of the transition relation $R$ and find all of those states that can be reached by a path in which each state is labeled with $f_i$. All such states should be labeled with $f$.

This step requires time $O(|S| + |R|)$.

The case in which $f = \EF \lor f_i$ is slightly more complicated and depends on the following observation.

**Lemma 1**: Let $M'$ be obtained from $M$ by deleting from $S$ all of those states at which $f_i$ does not hold and restricting $R$ and $L$ accordingly. Thus, $M' = (S', R', L')$ where $S' = \{ s \in S | M.s \models f_i \}$, $R' = R|_{S' \times S'}$, and $L' = L|_{S'}$. Then $M.s \models \EF f_i$ iff the following two conditions are satisfied:

1. $s \in S'$
2. there exists a path in $S'$ that leads from $s$ to some node $t$ in a non-trivial strongly connected component of the graph $(S', R')$.

**Proof**: Assume that $M.s \models \EF f_i$. Clearly $s \in S'$. Let $\pi$ be an infinite path starting at $s$ such that $f_i$ holds at each state on $\pi$. Since $M$ is finite, it must be possible to write $\pi$ as $\pi = \pi_1 \pi_2$ where $\pi_1$ is a finite initial segment and $\pi_2$ is an infinite suffix of $\pi$ with the property that each state on $\pi_2$ occurs...
infinitely often. Obviously \( \pi_0 \) is contained in \( S' \). Let \( C \) be the set of states in \( \pi_1 \). \( C \) is a nontrivial strongly connected component of \( S' \). To see this, let \( s_1 \) and \( s_2 \) be states in \( C \). Pick some instance of \( s_1 \) on \( \pi_1 \). By the way in which \( \pi_1 \) was selected, we know that there is an instance of \( s_1 \) further along \( \pi_1 \). The segment from \( s_1 \) to \( s_2 \) lies entirely within \( C \) and hence within \( S' \). This segment is a finite path from \( s_1 \) to \( s_2 \) in \( S' \). Thus, both condition (1) and condition (2) are satisfied.

Next, assume that conditions (1) and (2) are satisfied. Let \( \pi_1 \) be the path from \( s \) to \( t \). Let \( \pi_2 \) be a finite path of length at least one that leads from \( t \) back to \( s \). The existence of \( \pi_2 \) is guaranteed since \( C \) is a non-trivial strongly connected component. All of the states on the infinite path \( \pi = \pi_1 \pi_2^\omega \) satisfy \( f \). Since \( \pi \) is also a possible path starting at \( s \) in \( M \), we see that \( M, s \models EGf \). 

The algorithm for the case of \( f = \text{EG}f_0 \) follows directly from the lemma. We construct the restricted Kripke structure \( M' = (S', R', L') \) as described in the statement of the lemma. We partition the graph \( (S', R') \) into strongly connected components and find those states that belong to nontrivial components. We then work backwards using the converse of \( R \) and find all of those states that can be reached by a path in which each state is labeled with \( f_1 \). This step also requires time \( O(|S| + |R|) \).

In order to handle an arbitrary CTL formula \( f_0 \), we successively apply the state labeling algorithm to the subformulas of \( f_0 \), starting with the shortest, most deeply nested and work outward to include all of \( f_0 \). Since each pass takes time \( O(|S| + |R|) \) and since \( f_0 \) has \( \text{length}(f_0) \) different subformulas, the entire algorithm requires \( O(\text{length}(f_0) \cdot (|S| + |R|)) \).

**Theorem 2:** There is an algorithm for determining whether a CTL formula \( f_0 \) is true in state \( s \) of the structure \( M = (S, R, L) \) that runs in time \( O(\text{length}(f_0) \cdot (|S| + |R|)) \).

### 4. Fairness Constraints

In verifying concurrent systems, we are occasionally interested only in correctness along *fair* execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over an imperfect (or lossy) channel we may also wish to restrict the set of computation sequences; in this case the unfair execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver due to erratic behavior by the channel.

Roughly speaking, a fairness condition asserts that requests for service are granted "sufficiently often."
Different concepts of what constitutes a "request" and what "sufficiently often" should mean give rise to a variety of notions of fairness. Indeed, many different types of fairness and approaches to dealing with them have been proposed in the literature; we refer the reader to [Gabbay et al 80], [J amport 80], [Quelle & Sifakis 82], and [Lehmann et al 81] for more extensive treatments. The text by Francez [Francez 86] also gives an excellent survey of the various types of fairness.

In this section we will show how to extend the CTL model checking algorithm to handle a simple but fundamental type of fairness in which certain predicates must hold infinitely often along every fair path. ([Clarke et al 86a] shows how to handle a richer class of fairness constraints.) In this case it follows from [Emerson & Halpern 83] that correctness of fair executions cannot be expressed in CTL.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL\(^F\), has the same syntax as CTL. But a structure is now a 4-tuple \( M = (S, R, L, F) \) where \( S, R, L \) have the same meaning as in the case of CTL, and \( F \) is a collection of predicates on \( S, F \subseteq 2^S \). A path \( \pi \) is \( F\)-fair if the following condition holds: for each \( G \in F \), there are infinitely many states on \( \pi \) which satisfy predicate \( G \). CTL\(^F\) has exactly the same semantics as CTL except that all path quantifiers range over fair paths. The first step in checking CTL\(^F\) formulas is to determine the fair strongly connected components of the graph of \( M \). A strongly connected component is fair if it contains at least one state from each set in \( F \). Formally, let \( F = \{G_1, \ldots, G_k\} \) be a collection of subsets of \( S \). A strongly connected component \( C \) of the graph of \( M \) is fair iff for each \( G_i \) in \( F \), there is a state \( t_i \in (C \cap G_i) \).

**Lemma 3:** Given any finite structure \( M = (S, R, L, F) \) where \( F \) is a set of fairness constraints and a state \( s_0 \in S \), the following two conditions are equivalent:

1. There exists an \( F\)-fair path in \( M \) starting at \( s_0 \).
2. There exists a fair strongly connected component \( C \) of the graph of \( M \) such that there is a finite path from \( s_0 \) to a state \( t \in C \).

The proof is straightforward and is given in [Clarke et al 86a]. We next extend our model checking algorithm to CTL\(^F\). We introduce an additional proposition \( Q \), which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph associated with the structure and marking a component as fair if it contains at least one state from each \( G_i \) in \( F \). By the above lemma every state in a fair strongly connected component is the start of an infinite fair path. Thus, we label a state with \( Q \) iff there is a path from that state to some
node of a fair strongly connected component. As usual we design the algorithm so that after it terminates each state will be labeled with the subformulas of \( f_0 \) true in that state. We consider the two interesting cases where \( f \) is a subformula of \( f_0 \) and either \( f = F[f_0 \cup f] \) or \( f = FG_f \). We assume that the states have already been labeled with the immediate subformulas of \( f \) by an earlier stage of the algorithm.

1. \( f = F[f_0 \cup f] \): \( f \) is true in a state iff the CTL formula \( F[f_0 \cup (f \wedge Q)] \) is true in that state, and this can be determined using the CTL model checker. Again, state \( s \) is labeled with \( f \) iff \( f \) is true in that state.

2. \( f = FG_f \): To determine \( s \models FG_f \) we use the procedure described in section 3 to check \( s \models FG_f \wedge Q \) in the structure with the additional proposition \( Q \).

It is easy to see that the above algorithm runs in time \( O(length(f_0) \cdot (|S| + |R|)) \).

**Theorem 4:** There is an algorithm for determining whether a CTL \( ^{F} \) formula \( f_0 \) is true in state \( s \) of the structure \( M = (S, R, L, F) \) with \( F \) as the set of fairness constraints that runs in time \( O(length(f_0) \cdot (|S| + |R|)) \).

5. **An Example**

In this section we illustrate how the model checker can be used to verify a simple, but not entirely trivial, concurrent program. The example is a two process mutual exclusion program that was manually proved correct using linear temporal logic by Owicki and Lamport in [Owicki & Lamport 82]. The program, expressed in a variant of the CSP programming language [Hoare 73], is shown in Figure 5.1. In this version of CSP processes may have global variables (e.g. \( p1 \) and \( p2 \)), and assignments to such variables are assumed to be atomic. Since our verification technique can only be used to analyze finite state concurrent systems, we require that all variables be boolean and that all messages between processes be signals. Labels (e.g. \( NC1 \) and \( NC2 \)) are used to indicate that flow of control has reached a particular point in some process. In our example there are two processes \( S1 \) and \( S2 \), and each process has three code regions: a noncritical region \( NCi \) in which the process computes some data values that it wishes to share with the other process, a trying region \( Ti \) in which the process executes a protocol to obtain entry into the critical section, and a critical section \( CSi \) in which the process updates shared variables. To prevent a race condition that might result in unpredictable values being assigned to the shared variables, only one process is allowed to be in its critical section at any given time. Note that the two processes are different; hence this is not a symmetric solution to the mutual exclusion problem. When the CSP program is compiled, a state graph with 77 states is obtained. Although this is not an extremely large state machine, it would nevertheless be quite tedious for a human to debug.
We initially run the verifier without any fairness constraints—See Figure 5-2. We first check to see if both processes are ever in their critical regions at the same time. This property is succinctly expressed by the CTI formula $\text{EF}(\text{CS1} \land \text{CS2})$. The verifier rapidly determines that the formula is false—hence, the program does guarantee mutual exclusion. Time is measured in $1/60$ of a second. The first component measures user CPU time. The second component measures system CPU time. We next check for absence of deadlock. This is expressed by the formula $\text{AG(EF(\text{CS1} \lor \text{CS2}))}$. The verifier determines that this formula is satisfied; thus, from any state that is reachable from the initial state it is always possible to get to either $\text{CS1}$ or $\text{CS2}$.

Absence of starvation for process 1 is expressed by the formula $\text{AG}(T1 \rightarrow \text{AF CS1})$. This property is not satisfied without a fairness constraint. The reason is quite simple. When we build the global state graph for the program we do not make any assumptions about the relative speeds of the two processes. Thus, the second process can make any number of steps between steps of the first process. In fact, the second process can even run forever, thereby preventing the first process from ever making another step. We can rule out the second type of behavior by means of fairness constraints which require that each process be given a chance to execute infinitely often. In Figure 5-3 we restart the verifier with several fairness constraints that prevent either process from remaining forever at the same statement while enabled to make a step. Under these assumptions the first process will never starve. However, the possibility of starvation still exists for the second process.

A good solution to the mutual exclusion problem should not require that processes alternate entry into their critical regions: $\text{CS1, CS2, CS1, CS2, ...}$. In order to test that the algorithm given in Figure 5-1 does not require strict alternation, we check the formula $\text{AG}(\text{CS1} \rightarrow \text{A}[\text{CS1} \lor \text{A}[(\neg \text{CS1} \land \text{A}[(\neg \text{CS1} \lor \text{CS2})])])$. This formula asserts that if process 1 enters its critical section and subsequently leaves it, then it cannot enter it again until process 2 has entered its critical section. The verifier determines that the formula is false in less than a second. This example shows how the basic temporal operators, particularly the until operator, can be nested to express complicated timing properties.

Finally, the verifier has a counterexample feature (that is not shown in the transcripts). When this feature is enabled and the model checker determines that a formula is false, it will attempt to find a path in the state graph which demonstrates that the negation of the formula is true. For example, if the formula has the form $\text{AG}(f)$, our system will produce a path to a state in which $\neg f$ holds. For instance, when the verifier determines that the last formula above is false, it prints out an execution of
\[ s :: [ \\
\quad p1, p2: bool; \\
\quad NC1, NC2, T1, T2, T2a, CS1, CS2: label; \\
\quad S1, S2: process; \\
\quad S1 :: [ \\
\quad \quad p1 := false; \\
\quad \quad *[ \\
\quad \quad \quad true -> \\
\quad \quad \quad \quad <<NC1>> skip; -- noncritical section 1 \\
\quad \quad \quad \quad p1 := true; \\
\quad \quad \quad \quad <<T1>> *[ p2 -> skip]; \\
\quad \quad \quad \quad <<CS1>> skip; -- critical section 1 \\
\quad \quad \quad \quad p1 := false \\
\quad \quad \quad ]; \\
\quad \quad S2 :: [ \\
\quad \quad \quad p2 := false; \\
\quad \quad \quad *[ \\
\quad \quad \quad \quad true -> \\
\quad \quad \quad \quad <<NC2>> skip; -- noncritical section 2 \\
\quad \quad \quad \quad p2 := true; \\
\quad \quad \quad \quad <<T2>> *[ p1 -> p2 := false; \\
\quad \quad \quad \quad <<T2a>> *[ p1 -> skip ]; \\
\quad \quad \quad \quad p2 := true ]; \\
\quad \quad \quad \quad <<CS2>> skip; -- critical section 2 \\
\quad \quad \quad \quad p2 := false \\
\quad \quad \quad ]; \\
\quad ] \\
\] \\

Figure 5-1: Two process mutual exclusion program.
CTL MODEL CHECKER (C version 2.5)

$I = EF(CS1 \& CS2)$.  
The equation is FALSE.

time: (2 4)

$I = AG(EF(CS1 \mid CS2))$.  
The equation is TRUE.

time: (4 2)

$I = AG(T1 \rightarrow AF CS1)$.  
The equation is FALSE.

time: (17 12)

Figure 5-2: Transcript of model checker execution (without fairness constraint).

Fairness constraint: $\neg NC1$.  
Fairness constraint: $\neg NC2$.  
Fairness constraint: $\neg CS1$.  
Fairness constraint: $\neg CS2$.  
Fairness constraint: $\neg T1 \mid P2$.  
Fairness constraint: $\neg 12 \mid p1$.  
Fairness constraint: $\neg T2 \mid \neg p1 \mid T2a$.  
Fairness constraint: .

$I = AG(T1 \rightarrow AF CS1)$.  
The equation is TRUE.

time: (10 0)

$I = AG(T2 \rightarrow AF CS2)$.  
The equation is FALSE.

time: (29 9)

$I = AG( CS1 \rightarrow \exists[CS1 U (\neg CS1 \& \neg (CS1 U CS2)])]$.  
The equation is FALSE.

time: (38 17)

Figure 5-3: Transcript of model checker execution (with fairness constraint).
the mutual exclusion program in which process 1 enters its critical region, leaves, and reenters without process 2 entering its critical section in the meantime. This feature is quite useful for debugging purposes.

6. Other Approaches

Several papers have considered the model checking problem for linear temporal logic formulas. Let $M = (S, R, I)$ be a Kripke Structure with $s_0 \in S$, and let $Af$ 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 if $M, s_0 \models Af$. Notice that $M, s \models Af$ iff $M, s \models \neg E \neg f$. Consequently, it is sufficient to be able to check the truth of formulas of the form $Ef$ where $f$ is a restricted path formula. In general, this problem is PSPACE-complete [Sistla & Clarke 86]. Although the proof of this PSPACE-completeness result is beyond the scope of our survey, it is easy to see that the model checking problem is NP-hard for formulas of the form $Ef$ where $f$ is restricted path formula. We show that the directed Hamiltonian path problem is reducible to the problem of determining whether $M, s \models f$ where

- $M$ is a finite structure,
- $s$ is a state in $M$ and
- $f$ is the assertion (using atomic propositions $p_1, \ldots, p_n$):

$$f[Fp_1 \land \ldots \land Fp_n \land G(p_1 \rightarrow XG\neg p_1) \land \ldots \land G(p_n \rightarrow XG\neg p_n)].$$

Consider an arbitrary directed graph $G = (V, A)$ where $V = \{v_1, \ldots, v_n\}$. We obtain a structure from $G$ by making proposition $p_i$ hold at node $v_i$ and false at all other nodes (for $1 \leq i \leq n$), and by adding a source node $u_i$ from which all $v_i$ are accessible (but not vice versa) and a sink node $v_i$ which is accessible from all $v_i$ (but not vice versa). Formally, let the structure $M = (U, B, I)$ consist of

$$U = V \cup \{u_i, u_j\} \text{ where } u_i, u_j \not\in V;$$

$$B = A \cup \{(u_i, v) | v \in V\} \cup \{(v, u_i) | v \in V\} \cup \{(u_i, u_j)\}; \text{ and}$$

$L$ is an assignment of propositions to states such that

- $p_i$ is true in $v_i$ for $1 \leq i \leq n$
- $p_j$ is false in $v_i$ for $1 \leq i, j \leq n, i \neq j$
- $p_i$ is false in $u_i, u_j$ for $1 \leq i \leq n$
It is easy to see that $\mathcal{M}, u_{1} \models f$ iff there is a directed infinite path in $\mathcal{M}$ starting at $u_{1}$ which goes through all $v_{i} \in V$ exactly once and ends in the self loop through $u_{1}$. Note that the formula $f$ in the above construction has essentially the same size as the as the graph $G$. Suppose that the length of the formula to be checked was known to be much smaller than the size of the Kripke structure under consideration. Would the complexity still be high in this case? A careful analysis by Lichtenstein and Pnueli [Lichtenstein & Pnueli 85] showed that although the complexity is apparently exponential in the length of the formula, it is linear in the size of the global state graph. We briefly describe their results below.

Let $f$ be a restricted path formula. The closure of $f$, $\text{CI}(f)$, is the smallest set of formulas containing $f$ and satisfying:

- $\neg f_i \in \text{CI}(f)$ iff $f_i \in \text{CI}(f)$
- if $f_i \vee f_j \in \text{CI}(f)$, then $f_i, f_j \in \text{CI}(f)$
- if $Xf_i \in \text{CI}(f)$, then $f_i \in \text{CI}(f)$
- if $\neg Xf_i \in \text{CI}(f)$, then $\neg f_i \in \text{CI}(f)$
- if $f_i U f_j \in \text{CI}(f)$, then $f_i, f_j \in \text{CI}(f)$

It can be shown that the size of $\text{CI}(f)$ is $5 \cdot \text{length}(f)$.

An atom is a pair $A = (s_A, F_A)$ with $s_A \in S$ and $F_A \subseteq (\text{CL}(f) \cup AP$ such that:

- for each proposition $Q \in AP$, $Q \in F_A$ iff $Q \in \text{I}(s_A)$
- for every $f_i \in \text{CL}(f)$, $f_i \in F_A$ iff $\neg f_i \in F_A$
- for every $f_i, f_j \in \text{CL}(f)$, $f_i \vee f_j \in F_A$ iff $f_i$ or $f_j \in F_A$
- for every $\neg Xf_i \in \text{CL}(f)$, $\neg Xf_i \in F_A$ iff $\neg f_i \in F_A$
- for every $f_i, f_j \in \text{CL}(f)$, $f_i U f_j \in F_A$ iff $f_i \in F_A$ or $f_i, X(f_i U f_j) \in F_A$

Now, a graph $G$ is constructed with the set of atoms as the set of vertices. $(A,B)$ is an edge of $G$ iff $(s_A, s_B) \in R$ and for every formula $f_i$, if $Xf_i \in F_A$, then $f_i \in F_B$. An eventuality sequence is an infinite path $\pi$ in $G$ such that if $f_i U f_j \in F_A$ for some atom $A$ on $\pi$, then there exists an atom $B$, reachable from $A$ along $\pi$, such that $f_j \in F_B$.

**Lemma 5:** $\mathcal{M}, v \models F$ iff there exists an eventuality sequence starting at an atom $(v, F)$ such that $f \in F$. 
A non-trivial strongly connected component $C$ of the graph $G$ is said to be self-fulfilling iff for every atom $A$ in $C$ and for every $f, f' \in F_A$ there exists an atom $B$ in $C$ such that $f, f' \in F_B$.

**Lemma 6:** $M \models f$ iff there exists an atom $A = (s, F)$ in $G$ such that $f \in F$ and there exists a path in $G$ from $A$ to a self-fulfilling strongly connected component.

Lemma 6 be used as the basis for a linear temporal logic model checking algorithm. This algorithm has the time complexity $O(|S| + |R|) \cdot 2^{\text{length}(\mathcal{F})}$. Lichtenstein and Pnueli further showed how this basic algorithm could be extended to handle a number of different notions of fairness with essentially the same complexity.

An alternative approach due to Vardi and Wolper [Vardi & Wolper 86] exploits the close relationship between linear temporal logic formulas and Büchi automata. A Büchi automaton is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- $\Sigma$ is an alphabet.
- $S$ is a set of states.
- $\rho : S \times \Sigma \rightarrow 2^S$ is a non-deterministic transition function.
- $S_o \subseteq S$ is a set of initial states.
- $F \subseteq S$ is a set of designated states.

A run of $A$ on an infinite word $w = a_1a_2\ldots$ is a sequence $s_0, s_1, \ldots$ where $s_i \in S_o$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0, s_1, \ldots$ is accepting if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many $i$'s such that $s_i = s$. The infinite word $w$ is accepted by $A$ if there is an accepting run of $A$ over $w$. The set of infinite words accepted by $A$ is denoted $L(A)$. The following theorem is proved in [Vardi & Wolper 86].

**Lemma 7:** For every linear temporal formula $\mathcal{F}$, a Büchi automaton $A_f$ can be constructed, where $\Sigma = 2^{\text{AP}}$ and $|S| \leq 2^{\text{length}(\mathcal{F})}$, such that $L(A_f)$ is exactly the set of computations satisfying the formula $\mathcal{F}$.

A Kripke Structure $M = (S, R, I)$ with initial state $s_0 \in S$ can be viewed as a Büchi automaton $A_M = (\Sigma, S, \{s_0\}, \rho, S)$ where $\Sigma = 2^{\text{AP}}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = I(s)$. Note that any infinite run of this automaton is accepting. $L(A_M)$ is the set of computations of $A_M$. Thus, in order to determine whether $M \models \mathcal{F}$ it is sufficient to check whether $L(A_M) \cap L(\neg \mathcal{F})$ is empty. This can be
determined by an automata theoretic construction with essentially the same time complexity as the Pnueli Lichtenstein algorithm.

One of the expected advantages of using linear temporal logic is that fairness constraints can be handled directly. However, if fairness constraints are included as part of the specifications, the formulas that must be checked will in general be quite large. For instance, consider a fairness constraint which requires that progress be made from any state in the program. The formula that expresses this property is

\[ \forall s \in S \neg \text{G}(at s) \rightarrow \text{<rest of specification>} \]

which has size \(O(|S|)\). This problem was realized by Lichtenstein and Pnueli and by Vardi and Wolper. They in fact handle fairness by means of fairness constraints in a manner very similar to the way it is handled in [Clarke et al 86a]. Another problem with using linear temporal logic is that in general it is impossible to handle specifications which involve existential path quantifiers. Although it is possible to check simple formulas of the form \( Ff \) where \( f \) is a restricted path formula, it is not possible to check formulas like \( AG(FFf) \), which is used to express absence of deadlock in the example in section 5. Moreover, model checking for the full logic CTL* is no more difficult than for linear temporal logic as was shown by Emerson and Lei [Emerson & Lei 85].

**Theorem 8:** If we are given an algorithm \( M_{LTL} \) to solve the model checking problem for linear temporal logic, then we can construct an algorithm \( M_{CTL^*} \) for the full logic CTL* that has the same order of complexity as \( M_{LTL} \).

### 7. Applications

Sequential circuit verification is a natural application for the type of veriler discussed in this paper. Bochmann [Bochmann 82] was probably the first to realize the usefulness of temporal logic for describing the behavior of circuits. He verified an implementation of a self-timed arbiter using linear temporal logic and what he called "reachability analysis." The work of Malachi and Owicki [Malachi & Owicki 81] identified additional temporal operators required to express interesting properties of circuits and also gave specifications for a large class of modules used in self-timed circuits. Although these researchers contributed significantly toward developing an adequate notation for expressing the correctness of sequential circuits, the problem of mechanically verifying a circuit remained unsolved.

In [Mishra & Clarke 85] Clarke and Mishra showed how the FMC algorithm could be used to verify various temporal properties of asynchronous circuits. They developed a technique for extracting a state
graph directly from a wire-list description of the circuit (i.e., from a description of the circuit in terms of its components and their interconnections). The model checker was then used to show that state graph satisfied various specifications expressed in temporal logic. In this way they were able to determine that a self-timed queue element described in Seitz' chapter of Mead and Conway [Seitz 80] did not satisfy its specifications. Their work was later extended by Browne, Clarke, Dill, and Mishra [Browne et al 86] who showed, in general, how a mixed gate and switch level circuit simulator could be used to extract a state graph from a structural description of a sequential circuit. The basic simulation algorithm is shown in Figure 7-1. Circuits are usually designed under the assumption that certain input sequences and combinations will not occur. Their program exploits this observation to prevent a combinatorial explosion in the number of states that are generated, by allowing the user to specify a set of conditions under which the inputs can change.

(The procedure below uses a hash table that maps node value assignments to states. To construct the state machine, call this procedure on a node_value_assignment for the initial state.)

```
procedure BuildGraph(Node_value_assignment) return a state
begin
    if there is a state for the node_value_assignment already in the table then return the state;
    else
        Create a new state;
        Label state with nodes that have 1 values;
        Store state and node values together in hash table;
        for each possible input assignment do
            Combine current values for internal nodes and input assignment into a new node_value_assignment;
            Simulate one step to find a new node assignment;
            Call BuildGraph recursively on new node assignment;
            Add value returned by previous line to successors of current state;
        end
    end
end
```

Figure 7-1: Algorithm For Constructing Kripke Structure From Circuit

The circuit simulator in [Browne et al 86] used a unit-delay timing model in which the switching delays of all the transistors and gates are assumed to be equal. While a unit-delay model is satisfactory for synchronous circuits, it may not be appropriate for asynchronous circuits. In [Dill & Clarke 86] Dill and Clarke showed how Kripke structures could be extracted from a gate level description of a circuit under a model of circuit behavior that permitted arbitrary non-zero delays to be associated with the outputs of the gates. The basic idea behind their approach is quite simple. Consider an AND gate with
two inputs, \(x\) and \(y\), and a single output \(z\). Assume that the gate is in an unstable configuration with \(x\) low, \(y\) high, and \(z\) high. The Kripke structure for the circuit containing this gate will have a state corresponding to the unstable configuration as shown in Figure 7-2. The state will have a self-loop and a transition to another state representing a stable configuration in which the output is low. Fairness constraints, as described in Section 4, are used to insure that the system doesn’t remain in an unstable configuration forever. In the case of the AND gate, it is sufficient to require that infinitely often \(z = x \land y\).

\[\text{Figure 7-2: Kripke structure for unstable configuration of AND gate.}\]

In practice, the arbitrary delay model is much too conservative. Many circuits are “almost speed independent”: They do not appear to be correct under a pure arbitrary delay model, but would work given reasonable assumptions about the relationships between the delays. When the circuit designer has a great deal of control over the magnitudes of circuit delays, exploiting more detailed knowledge of circuit timing can result in smaller and faster circuits. In fact, actual circuits often rely on such assumptions. In [Browne et al 85] and [Dill 86] a method is described for adding such assumptions to a circuit description and incorporating them into the state-graph construction. Possible timing constraints include constant upper and lower bounds on individual delays, and bounds on the differences between delays. Using constraints of this form, one can say for example: “the delay of the first AND gate is between 5 and 10 nanoseconds” or “the delay of the first AND gate is greater than the delay of the second AND gate.” The state graph constructed with respect to a particular set of delay assumptions rules out some circuit executions which would be allowed under an arbitrary delay model. Hence, formulas in CTL which might not have been true in an arbitrary delay model may be true with respect to particular delay assumptions (because all the counterexample paths are ruled out by the delay assumptions). This technique was applied to a patented asynchronous queue cell in [Browne et al 85]. The authors determined that the circuit did not meet its specifications under the arbitrary element delay...
model. However, under the assumption that the input was slower than two of the circuit gates, they showed that the circuit met its temporal logic specifications.

An alternative approach obtains the state diagram by compilation from a specification of the original (synchronous) circuit in a simple programming language-like notation. Browne and Clarke ([Browne et al 86], [Browne & Clarke 86]) use a Pascal-like state machine description language called SML for this purpose. The language includes the standard control structures if, while, and loop/exit. A begin statement is also provided for simultaneous execution of statements in lock-step. Since SML programs will ultimately be implemented in hardware, the only data types permitted are boolean and (bounded) integer. The output of the SML compiler is a deterministic Moore Machine that can be automatically implemented as a PLA, PAL, or a ROM. The output can also be analyzed for correctness using the EMC algorithm. In [Browne 86] Browne describes a specialized version of the EMC algorithm that can check Moore machines much more rapidly than the original algorithm.

Another potential area of application is the verification of network communication protocols. The alternating bit protocol [Bartlet et al 69] for reliable transmission of messages by a noisy communication channel is a simple example of such an algorithm. By using the CTL model checking procedure it is possible to determine in a few seconds whether this protocol meets its specifications [Clarke et al 85]. Sifakis at Grenoble [Quielle & Sifakis 81] and Kurshan at Bell Labs [Kurshan 86] have also considered applications involving network protocols. The delay assumptions mentioned above may be useful for describing the real-time behavior of such protocols.

8. Conclusion

Although the verification technique described in this paper has already been used to find some nontrivial errors in circuit designs and communications protocols, more research needs to be done before it will become a truly practical debugging tool for use by system designers. One problem is the expressibility of the underlying temporal logic. For circuit specification timing diagrams may be more natural to use than temporal logic formulas. Of course, temporal logic is more general since there is no analogue of negation, disjunction, or conjunction for timing diagrams. It may be possible to either systematically translate timing diagrams into temporal logic formulas or check them directly using an algorithm similar to the one used by the model checker. If so, this would simplify the task of specifying a complicated circuit and also allow the designer to be more confident that specifications actually mean what he thinks they mean.
The most important problem, however, is the state explosion problem. There are several different strategies for handling this problem. In verifying asynchronous circuits, for example, buggy circuits sometimes result in much larger state graphs than correct circuits. This happens because the activity in the circuit is much more disordered after an error has occurred. One possible solution in this case is to run the program which builds the state graph and the model checker as co-routines, creating states only as they need to be referenced by the model-checker. In [Dill 86] this technique is called lazy state generation, by analogy to lazy evaluation in programming language implementations. By using this method, an error could be discovered and reported after constructing only a small part of the entire state graph; this would not only speed up the verification process, it would also make it possible to verify some circuits which could not be verified if the entire graph had to be constructed.

Another approach to the state explosion problem is to exploit the hierarchical structure of complex finite state concurrent systems. If an appropriate subset of CTL is used ([Mishra & Clarke 85], [Clarke et al 86b]), then lower level subcircuits can be simplified by "hiding" some of their internal nodes (more precisely, making it illegal to use them in temporal logic formulas) and merging groups of states that become indistinguishable into single state. Preliminary research in [Mishra & Clarke 85] indicates that by using this technique it may be possible to cut-down dramatically on the number of states that need to be examined.

Finally, special techniques may be appropriate for concurrent systems that are composed of many identical processes. Consider, for example, a distributed mutual exclusion algorithm for processes arranged in a ring network in which mutual exclusion is guaranteed by means of a token that is passed around the ring ([Dijkstra 85], [Kurshan 85], [Martin 85]). A strategy that is often used for debugging such systems is to consider first a reduced system with one or two processes. If it is possible to show that the reduced system is correct and if the individual processes are really identical, then one is tempted to conclude that the entire system will be correct. In [Clarke et al 86b] an attempt is made to provide a solid theoretical basis that will prevent fallacious conclusions in arguments of this type. The authors describe a temporal logic called Indexed CTL*, or ICTL*, for specifying networks of identical processes. The logic includes all of CTL* with the exception of the nexttime operator; in addition, it permits formulas of the form \( \bigwedge f(i) \) and \( \bigvee f(i) \) where \( f(i) \) is a formula in which all of the atomic propositions are subscripted by \( i \). A Kripke structure for a family of \( N \) identical processes may be obtained as a product of the state graphs of the individual processes. Instances of the same atomic proposition in different processes are distinguished by using the number of the process as a subscript, thus, \( i \), represents the instance of atomic proposition \( i \) associated with process \( S \).
Since a closed formula of the new logic cannot contain any atomic propositions with constant index values, it is impossible to refer to a specific process by writing such a formula. Hence, changing the number of processes in a family of identical processes should not affect the truth of a formula in the logic. This intuitive idea is made precise by introducing a new notion of \textit{bisimulation} [Milner 79] between two Kripke structures with the same set of indexed propositions but different sets of index values. It is possible to prove that if two structures correspond in this manner, a closed formula of Indexed CTL$^*$ will be true in the initial state of one if and only if it is true in the initial state of the other.

These ideas are illustrated in [Clarke et al 86b] by considering the distributed mutual exclusion algorithm mentioned above. The atomic proposition $c_i$ is true when the $i$-th process is in its critical region, and the atomic proposition $d_i$ is true when the $i$-th process is delayed waiting to enter its critical region. A typical requirement for such a system is that a process waiting to enter its critical region will eventually do so. This condition is easily expressed in ICTL$^*$ by the formula $\forall XG(d_i \Rightarrow \forall Fc_i)$. The results of [Clarke et al 86b] can be used to show that exactly the same ICTL$^*$ formulas hold in a network with 1000 processes as hold in a network with two processes. The EMC algorithm can be used to check automatically that the above formula holds in networks of size two and conclude that it will also hold in networks of size 1000. At present this methodology has only been partially automated, however. The bisimulation must be established by hand and this generally requires some representation of the larger Kripke structure. Several researchers are attempting to find a way of automating this phase in a manner that avoids building the larger Kripke structure.

Other techniques for avoiding the state explosion problem are being investigated by Kurshan and Wolper. In Kurshan’s system [Kurshan 85] this problem is handled by using a homomorphism to collapse a large state machine into a much smaller one while preserving those properties that are important for verification. Since Kurshan does not use temporal logic formulas for specification, he has no analogue of the indexed formulas or of the bisimulation theorem used in [Clarke et al 86b]. Wolper [Wolper 86] considers a logic somewhat like ICTL$^*$ for reasoning about programs that are data-independent; however, his indexed variables range over data elements, not over processes. Also, there is no notion of correspondence between structures in his work. Some ultimate limitations on this type of reasoning are discussed in Apt and Kozen [Apt & Kozen 86].
References


END

FILMED

MARCH, 1988

DTIC