Using Temporal Logic for Automatic Verification of Finite State Systems

E. M. Clarke
Carnegie Mellon University

M. C. Browne
Carnegie Mellon University

E. A. Emerson
University of Texas, Austin

A. P. Sistla
University of Massachusetts, Amherst

1. Introduction.

Temporal logic has been extensively investigated for proving properties of programs—particularly for programs that involve nondeterminism or concurrency ([9], [11], [12]). However, most of the verification techniques developed so far involve manual construction of proofs, a task that may require a good deal of ingenuity and is usually quite tedious. In a series of papers ([1], [5], [6], [10]) we have argued that proof construction is unnecessary in the case of finite state systems and can be replaced by a model theoretic approach which will mechanically determine if the system meets a specification expressed in a propositional temporal logic. In this paper we survey that work and give a detailed example of how our approach might be used in verifying a finite state hardware controller.

The basic idea behind our approach is quite simple. The state transition graph of a finite state system can be viewed as a finite Kripke structure, and an efficient algorithm can be given to determine whether a structure is a model of a particular formula—i.e., to determine if the program meets its specification. The algorithm, which we call a model checker, is similar to the global flow analysis algorithms used in compiler optimization and has complexity linear in both the size of the structure and the size of the specification. When the number of states is not excessive (i.e. not more than a few thousand) we believe that our technique may provide a powerful debugging tool.

Since our specification language is a branching-time temporal logic, it follows from ([7], [8]) that our logic cannot, in general, express correctness of fair execution sequences. The alternative of using a linear time logic is ruled out because any model checker for such a logic must have high

This research was supported by NSF Grants MCS-815553 and MCS-8302878.
complexity ([15]). We overcome this problem by moving fairness requirements into the semantics of our logic. Specifically, we change the definition of our basic modalities so that only fair paths are considered. Our previous model checking algorithm is modified to handle this extended logic without changing its complexity.

An obvious application for our method is in verifying complicated finite state systems that will ultimately be implemented as sequential circuits. Although this has been important problem for a long time, lack of any formal and efficient method of verification has prevented the creation of practical design aids. Since all the known techniques of simulation and prototype testing are time-consuming and not very reliable, there is an acute need for such tools. We illustrate our approach to this problem by verifying the correctness of a moderately tricky traffic controller expressed in a high-level state machine description language with a Pascal-like syntax (called SML). The output of the SML compiler can also be used to generate a PLA, PAL, or ROM—thus, permitting state machines that have been verified by our techniques to be implemented as circuits.

Most prior research on verifying finite state systems has involved some type of state reachability analysis. For example, in [16] and [18] reachability techniques are described for detection of system deadlocks, unspecified message receptions. and non-executable process interactions. An obvious advantage that our approach has over such methods is flexibility; our use of temporal logic provides a uniform notation for expressing a wide variety of correctness properties. Furthermore, it is unnecessary to formulate all of the specifications as reachability assertions since the model checker can handle both safety and liveness properties with equal facility.

Perhaps the research that is most closely related to our own is that of Queille and Sifakis ([13], [14]), who have independently developed a system which will automatically check that a finite state CSP program satisfies a specification in temporal logic. The logical system that is used in [13], is not as expressive as our logic, however, and no attempt is made to handle fairness properties. Although fairness is discussed in [14], the approach that is used is much different from the one that we have adopted. Special temporal operators are introduced for asserting that a property must hold on fair paths, but neither a complexity analysis nor an efficient model checking algorithm is given for the extended logic.

Our paper is organized as follows. Section 2 contains the syntax and semantics of our logic. In Section 3 fixpoint characterizations are given for the various temporal operators. The fixpoint characterization are used in Section 4 to develop the basic model checking algorithm. An extension of the algorithm which only considers fair computations is discussed in section 5. In section 6 we
outline how a model can be extracted from a program in a high-level state machine description language with a Pascal-like syntax and illustrate its use with examples. The paper concludes in section 7 with a discussion of directions for future research including the possibility of making our approach hierarchical.

2. The Specification Language.

Our specification language is a propositional, branching-time temporal logic called Computation Tree Logic (CTL) and is similar to the logical systems described in [3], [5], and [7]. The formal syntax for CTL is given below. AP is the underlying set of atomic propositions.

1. Every atomic proposition \( p \in \text{AP} \) is a CTL formula.

2. If \( f_1 \) and \( f_2 \) are CTL formulas, then so are \( \neg f_1, f_1 \land f_2, AXf_1, EXf_1, A[f_1 \cup f_2], \) and \( E[f_1 \cup f_2] \).

The symbols \( \land \) and \( \neg \) have their usual meanings. \( X \) is the nexttime operator; the formula \( AXf_1 \) \( (EXf_1) \) intuitively means that \( f_1 \) holds in every (in some) immediate successor of the current program state. \( U \) is the until operator; the formula \( A[f_1 \cup f_2] \) \( (E[f_1 \cup f_2]) \) intuitively means that for every computation path (for some computation path), there exists an initial prefix of the path such that \( f_2 \) holds at the last state of the prefix and \( f_1 \) holds at all other states along the prefix.

We define the semantics of CTL formulas with respect to a labeled state-transition graph (or Kripke structure). Formally, a CTL structure is a triple \( M = (S, R, P) \) where

1. \( S \) is a finite set of states.

2. \( R \) is a binary relation on \( S \) \( (R \subseteq S \times S) \) which gives the possible transitions between states and must be total, i.e. \( \forall x \in S \exists y \in S [(x,y) \in R] \).

3. \( P : S \rightarrow 2^\text{AP} \) assigns to each state the set of atomic propositions true in that state.

A path is an infinite sequence of states \( (s_0, s_1, s_2, \ldots) \) such that \( \forall i [(s_i, s_{i+1}) \in R] \). For any structure \( M = (S, R, P) \) and state \( s_0 \in S \), there is an infinite computation tree with root labeled \( s_0 \) such that \( s \rightarrow t \) is an arc in the tree iff \( (s, t) \in R \). Figure 2-1 shows a CTL structure and the associated computation tree rooted at \( s_0 \).

We use the standard notation to indicate truth in a structure: \( M, s_0 \models f \) means that formula \( f \) holds at state \( s_0 \) in structure \( M \). When the structure \( M \) is understood, we simply write \( s_0 \models f \). The relation \( \models \) is defined inductively as follows:

\[ s_0 \models p \quad \text{iff} \quad p \in P(s_0). \]
Figure 2-1: A CTL structure and its computation tree
$s_0 \models \neg f$  \hspace{1cm} \text{iff} \hspace{1cm} \neg(s_0 \models f).

$s_0 \models f_1 \land f_2$  \hspace{1cm} \text{iff} \hspace{1cm} $s_0 \models f_1$, and $s_0 \models f_2$.

$s_0 \models \text{AX} f_1$  \hspace{1cm} \text{iff} \hspace{1cm} \text{for all states } t \text{ such that } (s_0, t) \in R, t \models f_1.$

$s_0 \models \text{EX} f_1$  \hspace{1cm} \text{iff} \hspace{1cm} \text{for some state } t \text{ such that } (s_0, t) \in R, t \models f_1.$

$s_0 \models A[f_1 \cup f_2]$  \hspace{1cm} \text{iff} \hspace{1cm} \text{for all paths } (s_0, s_1, \ldots). \exists (i \geq 0 \land s_i \models f_2 \land \forall [0 \leq j < i \land s_j \models f_1]).$

$s_0 \models E [f_1 \cup f_2]$  \hspace{1cm} \text{iff} \hspace{1cm} \text{for some path } (s_0, s_1, \ldots). \exists (i \geq 0 \land s_i \models f_2 \land \forall [0 \leq j < i \land s_j \models f_1]).$

We will also use the following abbreviations in writing CTL formulas:

$f_1 \lor f_2 = \neg(\neg f_1 \land \neg f_2)$, $f_1 \rightarrow f_2 = \neg f_1 \lor f_2$, and $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \land (f_2 \rightarrow f_1)$ for logical disjunction, implication, and equivalence, respectively.

$AF(f) = A[\text{True} \cup f]$ intuitively means that $f$ holds in the future along every path from $s_0$, i.e., $f$ is inevitable.

$EF(f) = E[\text{True} \cup f]$ means that there is some path from $s_0$ that leads to a state at which $f$ holds, i.e., $f$ potentially holds.

$AG(f) = -EF(\neg f)$ means that $f$ holds at every state on every path from $s_0$, i.e., $f$ holds globally.

$EG(f) = -AF(\neg f)$ means that there is some path from $s_0$ on which $f$ holds at every state.

3. Fixpoint Characterizations

Each of the modal operators such as AU, EG, EF, etc., may be characterized as an extremal fixpoint of an appropriate monotonic functional. Let $M = (S, R, P)$ be an arbitrary structure. We use $\text{PRED}(S)$ to denote the lattice of total predicates over $S$ where each predicate is identified with the set of states that make it true and the ordering is set inclusion. Thus, each formula $f$ is associated with an element of $\text{PRED}(S)$, namely $\{s : M, s = f\}$. Let $\tau : \text{PRED}(S) \rightarrow \text{PRED}(S)$ be given; then

1. $\tau$ is monotonic provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$;

2. $\tau$ is u-continuous provided that $P_1 \subseteq P_2 \subseteq \ldots$ implies $\tau(\cup_i P_i) = \cup_i \tau(P_i)$;

3. $\tau$ is $\wedge$-continuous provided that $P_1 \supseteq P_2 \supseteq \ldots$ implies $\tau(\cap_i P_i) = \cap_i \tau(P_i)$.

A monotonic functional $\tau$ on $\text{PRED}(S)$ always has both a least fixpoint, $\text{lfP} \tau[Z]$, and a greatest fixpoint, $\text{gfp} \tau[Z]$ (see Tarski [1]). $\text{lfP} \tau[Z] = \neg (Z \cap \tau[Z] = Z)$ whenever $\tau$ is monotonic, and $\text{lfP} \tau[Z] = \cup \tau[\text{False}]$ whenever $\tau$ is also u-continuous. $\text{gfp} \tau[Z] = \cap \{Z \cap \tau[Z] = Z\}$ whenever $\tau$ is monotonic, and $\text{gfp} \tau[Z] = \wedge \tau[\text{True}]$ whenever $\tau$ is also $\wedge$-continuous.
The modal operators have the following fixpoint characterizations:

\[ A[f_1, U f_2] = \text{Ifp}Z.f_2 \lor (f_1 \land AXZ) \]
\[ E[f_1, U f_2] = \text{Ifp}Z.f_2 \lor (f_1 \land EXZ) \]
\[ AFI_1 = \text{Ifp}Z.f_1 \lor AXZ \]
\[ EFI_1 = \text{Ifp}Z.f_1 \lor EXZ \]
\[ AGI_1 = \text{gfp}Z.f_1 \land AXZ \]
\[ EGI_1 = \text{gfp}Z.f_1 \land EXZ \]

If there is an upper bound on the branching degree of each node in computation tree corresponding to relation \( R \) and initial state \( s_0 \), then each of the functionals used in the fixpoint characterizations above is \( \cup \)-continuous and \( \land \)-continuous as well as monotonic. This will, of course, be the case for all relations considered in this paper. We show that the fixpoint characterization for \( EF \) is correct:

**Lemma 1:** \( EFi_1 \) is the least fixpoint of the functional \( \tau[Z] = f_1 \lor EXZ \).

**Proof.** We first show that \( EFi_1 \) is a fixpoint of \( \tau[Z] \): Suppose \( s_0 \models EFi_1 \). Then by definition of \( \models \), there is a path \( (s_0, s_1, s_2, \ldots) \) in \( M \) such that for some \( k \), \( s_k \models f_1 \). If \( k = 0 \), \( s_0 \models f_1 \). Otherwise \( s_1 \models EFi_1 \) and \( s_0 \models EEXFi_1 \). Thus, \( EFi_1 \subseteq f_1 \lor EEXFi_1 \). Similarly, if \( s_0 \models f_1 \lor EEXFi_1 \), then \( s_0 \models f_1 \) or \( s_0 \models EEXFi_1 \). In either case, \( s_0 \models EFi_1 \) and \( f_1 \lor EEXFi_1 \subseteq EFi_1 \). Thus \( EFi_1 = f_1 \lor EEXFi_1 \).

To see that \( EFi_1 \) is the least fixpoint of \( \tau[Z] \), it suffices to show that \( EFi_1 = \bigcup_{i \geq 0} \tau_i[\text{False}] \). It follows by a straightforward induction on \( i \) that \( s_0 \in \tau_i[\text{False}] \) iff there is a finite path \( (s_0, s_1, \ldots, s_i) \) in \( M \) and a \( j \leq i \) for which \( s_j \models f_1 \). \( \square \)

**4. Model Checker**

Assume that we wish to determine whether formula \( f \) is true in the finite structure \( M = (S, R, P) \). Let \( \text{sub}^+(f_0) \) denote the set of subformulas of \( f_0 \) with main connective other than \( \neg \). We label each state \( s \in S \) with the set of positive/negative formulas \( f \) in \( \text{sub}^+(f_0) \) so that \( f \in \text{label}(s) \) iff \( M, s \models f \) and \( \neg f \notin \text{label}(s) \) iff \( M, s \models \neg f \).

The algorithm makes \( n + 1 \) passes where \( n = \text{length}(f_0) \). On pass \( i \), every state \( s \in S \) is labelled with \( f \) or \( \neg f \) for each formula \( f \in \text{sub}^+(f_0) \) of length \( i \). Information gathered in earlier passes about formulas of length less than \( i \) is used to perform the labelling. For example, if \( f = f_1 \land f_2 \), then \( f \) should be
placed in the set for s precisely when f₁ and f₂ are already present in the set for s. For modalities such as A[f₁, U f₂] information from the successor states of s (as well as from s itself) is used. Since A[f₁, U f₂] = f₂ ∨ (∀v. AX A[f₁, U f₂]). A[f₁, U f₂] should be placed in the set for s when f₂ is already in the set for s or when f₁ is in the set for s and A[f₁, U f₂] is in the set of each immediate successor state of s.

Satisfaction of A[f₁, U f₂] may be seen to "radiate" outward from states where it holds immediately by virtue of f₂ holding. Let

\[
\begin{align*}
A[f₁, U f₂]₀ &= f₂, \\
A[f₁, U f₂]_k &= f₁ ∧ AX(A[f₁, U f₂])_k.
\end{align*}
\]

It can be shown that M,s ⊨ (A[f₁, U f₂])ₖ if M,s ⊨ A[f₁, U f₂] and along every path starting at s, f₂ holds by the kth state following s. Thus, states where (A[f₁, U f₂])₀ holds are found first, then states where (A[f₁, U f₂])₁ holds, etc. If A[f₁, U f₂] holds, A[f₁, U f₂]^[card(S)] must hold since all loop-free paths in M are of length ≤ card(S). Thus, if after card(S) steps of radiating outward A[f₁, U f₂] has still not been found to hold at state s, then put ¬A[f₁, U f₂] in the set for s.

The algorithm for pass i is given in Figure 4-1. Figure 4-2 - 4-6 give snapshots of the algorithm in operation on the structure shown for the formula AFb ∧ EFa (which abbreviates AFb ∧ ¬AF¬a).

The algorithm presented above runs in time O(length(f) · (card(S) + card(R)))². In [6] we describe a more efficient model checking algorithm that is linear in the size of the CTL structure. This algorithm is also fairly simple, since it is based on a depth-first-search of the CTL structure.

**Theorem 2:** There is an algorithm for determining whether a CTL formula f is true in state s of the structure M = (S, R, P) which runs in time O(length(f) : (card(S) + card(R))). □

The only remaining feature of the verifier that we need to discuss is the counterexample feature. When the model checker determines that a formula is false, it will attempt to find a path in the graph which demonstrates that the negation of the formula is true. For instance, if the formula has the form AG(f), our system will produce a path to a state in which ¬f holds. This feature is quite useful for debugging purposes.

The verifier has been operational since January, 1982. It is written in C and runs on a VAX 11/780 under Unix. Several programs are written for generating state machines for analysis by the model checker. One will be described in section 6. Another generates state machines from a switch level
for every state \( s \in S \) do
    for every \( f \in \text{sub}^i(f_0) \) of length \( i \) do
        if \( f = A[f_1Uf_2] \) and \( f_1 \in \text{set}(s) \) or
            \( f = E[f_1Uf_2] \) and \( f_1 \in \text{set}(s) \) or
            \( f = Ef_i \) and \( \exists t[(s,t) \in R \text{ and } f_i \in \text{set}(t)] \) or
            \( f = f_1 \land f_2 \) and \( f_1 \in \text{set}(s) \) and \( f_2 \in \text{set}(s) \)
            then add \( f \) to \( \text{set}(s) \) end if
    end for
end for;

A: for \( j = 1 \) to \( \text{card}(s) \) do
    for every state \( s \in S \) do
        for every \( f \in \text{sub}^i(f_0) \) of length \( i \) do
            if \( f = A[f_1Uf_2] \) and \( f_1 \in \text{set}(s) \) and \( \forall t[(s,t) \in R \rightarrow f_i \in \text{set}(t)] \) or
                \( f = E[f_1Uf_2] \) and \( f_1 \in \text{set}(s) \) and \( \exists t[(s,t) \in R \land f_i \in \text{set}(t)] \)
                then add \( f \) to \( \text{set}(s) \) end if
        end for
    end for
B: end for
end for;

for every state \( s \in S \) do
    for every \( f \in \text{sub}^i(f_0) \) of length \( i \) do
        if \( f \in \text{set}(s) \) then add \( \neg f \) to \( \text{set}(s) \) end if
    end for
C: end for

Figure 4.1: Pass \( i \) of Model Checking Algorithm
Figure 4.2: First time at label A in Pass 1
Figure 4.3: First time at label B in Pass 1
Figure 4.4: Second time at label B in Pass 1
Figure 4.5: First time at label C in Pass 1
Figure 4.6: At termination
circuit description by using a unit-delay simulator to discover the transition relation of the corresponding CTL structure. The third is a compiler for a finite state subset of the CSP programming language.

5. Introducing Fairness into CTL

Occasionally, we are only interested in the correctness of fair execution sequences. For example, we may wish to consider only execution sequences in which each process is executed infinitely often. Unfortunately, this type of property cannot be expressed directly in CTL ([7], [8]). The alternative of using a linear time logic is ruled out because any model checker for such a logic must have high complexity. In [15] we prove that the following problem is NP-complete for linear temporal logic with the $F$ and $G$ operators and PSPACE-complete for linear temporal logic with the $U$ operator or with the $F$ and $X$ operators:

Given a structure $M = (S, R, P)$, a state $s_0 \in S$ and a formula $f$, is there a path $p$ in $M$ starting from $s_0$ such that $p, s_0 \models f$?

In order to handle such properties we must modify the semantics of CTL slightly. Initially, the model checker will prompt the user for a series of fairness constraints. Each constraint can be an arbitrary formula of the logic. A path is said to be fair with respect to a set of fairness constraints if each constraint holds infinitely often along the path. More formally, a structure is a 4-tuple $(S, R, P, F)$ where $S$, $R$, $P$ have the same meaning as in section 2, and $F$ is a collection of predicates on $S$ i.e. $F \subseteq 2^S$. A path $p$ is $F$-fair iff the following condition holds: for each $G \in F$, there are infinitely many states on $p$ which satisfy predicate $g$. The definition in section 2 of truth in a CTL structure is now modified so that the path quantifiers in CTL formulas are restricted to $F$-fair paths. Examples of fairness constraints can be found in section 6. In [6] we show that handling fairness in this manner does not change the linear time complexity of the model checker. We will not discuss in this paper the specific modifications that must be made in the basic model checking algorithm; however, the key observation is contained in the following lemma.

Lemma 3: Given any finite structure $M = (S, R, P)$, collection $F = \{G_1, \ldots, G_k\}$ of subsets of $S$, and 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 strongly connected component $C$ of (the graph of) $M$ such that
   a. there is a finite path from $s_0$ to a state $t \in C$, and
   b. for each $G_i$, there is a state $t_i \in C \cap G_i$. 
Proof:

(1) $\Rightarrow$ (2): Suppose the F-fair path $s_0, s_1, s_2, \ldots$ exists in $M$. Then for each $G_i$ there is a state $t_i \in G_i$ for which there exist infinitely many $s_i$ that are equal to $t_i$. So for each pair $t_i, t_j$ there is a path (which is some finite segment of the original path) from $t_i$ to $t_j$. It follows that all the $t_i$ lie in the same strongly connected component $C$ of $M$. Certainly, there is a path from $s_0$ to some node $t \in C$ (take $t = t_1$). Moreover, by the choice of the $t_i$, each $t_i \in C \cap G_i$. Thus, $C$ is the desired strongly connected component of (2).

(2) $\Rightarrow$ (1): Suppose the strongly connected component $C$ exists in $M$. Then finite paths of the following forms are also present in $M$: $(s_0, \ldots, t_i), (t_i, \ldots, t_2), \ldots (t_{k-1}, \ldots, t_k)$, and $(t_k, \ldots, t_1)$. We then concatenate these finite paths to get a path: $s_0, \ldots, t_i, \ldots, t_2, \ldots, t_{k-1}, \ldots, t_1, \ldots, t_k, \ldots, t_i, \ldots, t_2, \ldots, t_{k-1}, \ldots, t_1, \ldots, t_k, \ldots$. This path starts at $s_0$, and for each $i$ there are infinitely many occurrences of $t_i \in G_i$ along it. Thus, this path is F-fair. □

6. Using the Model Checker to Verify a Finite State Hardware Controller

In order to assist with the design and verification of finite state machines, we have designed a language named SML (state machine language). In addition to being useful for verification, SML also provides a succinct notation for describing complicated finite state machines. A program written in SML is compiled into a finite state machine, which can then be verified using the model checker or implemented in hardware. At CMU, we have implemented an SML compiler that runs on a VAX 11/780. We also have access to design tools that can implement a finite state machine produced by the compiler as either a ROM, a PLA, or a PAL.

6.1. The Description Language and its Semantics

An SML program represents a synchronous circuit that implements a Moore machine. At a clock transition, the program examines its input signals and changes its internal state and output signals accordingly. Since we are dealing with digital circuits where wires are either high or low, the major data type is boolean. Each boolean variable may be declared to be either an input changed only by the external world but visible to the program, an output changed only by the program but visible to the external world, or an internal changed and seen only by the program. The hardware implementation of boolean variables may also be declared to be either active high or active low. The use of mixed logic in SML is permitted. Internal integer variables are also provided.

SML programs are similar in appearance to many imperative programming languages. SML statements include if, while, and loop/exit. A cobegin is provided to allow several statements to execute concurrently in lockstep. There is also a simple macro facility.
The semantics of SML programs is different from most programming languages, since we are not only interested in what a statement does, but how much time it takes to do it. In this respect, SML was influenced by the semantics of ESTEREL [2]. The complete semantics for SML will not be given here but will appear in a forthcoming paper [4]. A program state is an ordered pair \( \langle S, \sigma \rangle \), consisting of a statement \( S \) and a function \( \sigma \) that gives values to all of the identifiers. The semantics consist of a set of rewrite rules that describe how a program state is transformed into a new program state. Each rewrite rule also specifies whether it takes a clock cycle to make the transformation or not. For example, two typical rewrite rules are:

\[
\langle \text{raise (I)}; S, \sigma \rangle \xrightarrow{t} \langle S, \sigma' \rangle
\]

where \( \sigma' = \sigma[I \leftarrow \text{true}] \)

\[
E = \text{false} \\
\langle \text{if } E \text{ then } S_1 \text{ endif}; S_2, \sigma \rangle \xrightarrow{0} \langle S_2, \sigma \rangle
\]

The first rule states that a raise statement followed by an arbitrary statement \( S \) can be rewritten in one clock cycle to statement \( S \) while simultaneously changing \( \sigma \) so that \( \sigma'(I) = \text{true} \). The second rule states that an if statement followed by an arbitrary statement \( S_2 \) can be rewritten in no time to statement \( S_2 \) if the condition is false.

Given any program state, we can repeatedly apply the rewrite rules to find a new state that can be reached in one clock cycle. This new state is a successor of the original state in the finite state machine. So starting from the initial program state (which consists of the entire program and a function which assigns 0 to all integers and \( \text{false} \) to all booleans), we can repeatedly find successor states until we have built the entire finite state machine.

6.2. Example: A Traffic Controller

The best way to illustrate the use of SML is by an example. We will use SML to design a traffic controller that is stationed at the intersection of a two-way highway going north and south and a one-way road going east. For the sake of simplicity, no turns are permitted. At the north, south, and east of this intersection, there is a sensor that goes high for at least one clock cycle when a car arrives. When the intersection is clear of cross traffic, the controller should raise a signal indicating that the car is permitted to cross the intersection. Once the car has crossed, the sensor that indicated the arrival of the car will go low.

Let the names of the sensors be \( N \) (north), \( S \) (south), and \( E \) (east). Furthermore, let \( N\text{-Go}, S\text{-Go}, \) and \( E\text{-Go} \) be the names of the output signals for each end of the intersection.
Now that the problem is defined, we can express the correctness conditions of the controller in CTL.

\[ AG \neg (E \cdot Go \land (N \cdot Go \lor S \cdot Go)) \]

This formula is a safety property that is true if the controller does not permit collisions to occur. There are also several interesting liveness properties:

\[ AG (\neg N \cdot Go \land N \rightarrow AF N \cdot Go) \]
\[ AG (\neg S \cdot Go \land S \rightarrow AF S \cdot Go) \]
\[ AG (\neg E \cdot Go \land E \rightarrow AF E \cdot Go) \]

These formulas state that every request to enter the intersection is eventually answered, so the controller is starvation-free. If all three of these formulas are true, the controller is deadlock-free as well.

\[ EF (N \cdot Go \land S \cdot Go) \]

Since we want to maximize the amount of traffic, this formula insures that the controller allows north and south traffic to cross the intersection simultaneously.

In addition to specifying the desired behavior of the controller, we must also specify the behavior of the cars. In particular, we don't want a car to enter the intersection and stay there forever. Since the model checker allows the specification of fairness constraints that must be true infinitely often, we must rephrase this condition to be that the cars must be out of the intersection infinitely often. Since a car from the north is in the intersection if \( N \cdot Go \) is true, and it stays there while \( N \) is true, the fairness constraint for cars from the north is \( \neg (N \cdot Go \land N) \). There are similar constraints for traffic from the south and east.

6.3. An implementation of the Traffic Controller in SML

One approach to this problem is to provide two locks: \( NS \cdot Lock \), which is true when north-south traffic is in the intersection, and \( EW \cdot Lock \), which is true when east-west traffic is in the intersection. Traffic from one direction is forbidden to enter the intersection if the lock in the other direction is true.

Figure 6.1 shows a program that uses this idea. The numbers at the beginning of each line were added for easy reference and are not part of the language.

A few comments are necessary to explain the operation of this program.

**Line 5:** In addition to declaring the two locks, \( N \cdot Req \), \( S \cdot Req \), and \( E \cdot Req \) are also declared to be internal. \( N \cdot Req \) will go high when a car arrives at the intersection from the north and go low when the car has crossed the intersection. \( S \cdot Req \) and \( E \cdot Req \) are similar.

**Lines 7-9:** Wait is a macro definition that delays until its parameter becomes true.
Figure 6-1: A First Attempt at Writing a Traffic Controller in SML

Line 12: If a car is not at the north end of the intersection (!N-Req), and the sensor at the north goes high (N), there is now a car at the north end of the intersection, so assert N-Req.

Lines 14 and 16: These statements do the same as line 12 for cars from the south and east.

Lines 18-31: This statement controls traffic from the north. The procedure is to lock the intersection (line 20), wait until the cross traffic releases the intersection (line 21), and then go (line 22). After the car has crossed (line 23), release the intersection if there is no south traffic about to enter the intersection (!S-Go & !S-Req) or if there is south traffic simultaneously leaving the
intersection (S-Go & !S) (line 25). Do not accept another request from the north until any east traffic finishes crossing (line 29).

**Lines 33-46:** This statement controls traffic from the south. The algorithm is the same as for north traffic, except that north traffic changes NS-Lock if both north traffic and south traffic want to change it simultaneously. On line 35, south traffic sets NS-Lock only if north traffic isn't about to enter the intersection and set it. On line 40, north traffic will release NS-Lock if it is leaving the intersection simultaneously, so it is not necessary to test (N-Go & !N).

**Lines 48-55:** This statement controls traffic from the east. Once there is no north-south traffic (line 50), the intersection is locked and the car is allowed to go (line 51). After the car leaves (line 52), the intersection is released.

This program was compiled into a 72 state machine in approximately 10 seconds of CPU time on a VAX. However, the transitions of this state machine are dependent on the state of the input. In order to remove this dependency, each state had to be replaced with 8 states, one for each possible combination of inputs. An additional 35 seconds of CPU time was required to convert this state machine into a 576 state machine that the model checker can handle. We have already developed a new model checker algorithm that circumvents this problem and we hope to implement it in the near future.

### 6.4. Verifying the Traffic Controller with the Model Checker

Figure 6-2 shows a transcript of the model checker running on the program in figure 6-1. The numbers in parentheses are the total user cpu time and "system time", in 1/60ths of a second. As the transcript shows, the program allows simultaneous north and south traffic and is collision-free, but it is not deadlock-free. The model checker provides a counter example that can be used to diagnose the problem. In state 390, cars from the north and the south are in the intersection, and there is a car waiting from the east. Furthermore, the car from the north is leaving the intersection (N is false), so the controller will not allow another car from the north to cross until the car from the east has crossed. In state 417, another car arrives from the north, so N-Req is raised in state 432. In state 432, the car from the south leaves the intersection (S is false). But since N-Req is high, the controller does not lower NS-Lock in state 523! So state 523 is a deadlock, where the car from the east is waiting for the north-south traffic to unlock the intersection, and the north-south traffic is waiting for the car from the east to cross the intersection.

As the counter example illustrates, the problem with the program in figure 6-1 is that a car from the south will not lower NS-Lock when it leaves the intersection if N-Req is high, since it expects a car from the north to enter the intersection. However, the car from the north might be waiting for a car
Figure 6-2: Verifying the First Traffic Controller Program

from the east to cross (line 29), so it will not enter, and a deadlock will result. A simple solution is to replace the wait at line 29 with a loop that will lower NS-Lock if south traffic leaves the intersection while east and north traffic is waiting. The wait at line 44 must also be replaced by a similar loop. The result of these changes is the program shown in figure 6-3. This program compiles into 69 states (552 states for the model checker) The correctness of this program is shown by the transcript in figure 6-4.
program intersect;

input N, S, E;
output N-Go, S-Go, E-Go;
internal NS-Lock, E-Lock, N-Req, S-Req, E-Req;

procedure wait [expr]
while ![expr] do endwhile
endproc

cobegin
  loop if N-Req & N then raise (N-Req) endif endloop
  loop if S-Req & S then raise (S-Req) endif endloop
  loop if E-Req & E then raise (E-Req) endif endloop

loop
  if N-Req then
    raise (NS-Lock);
    wait (!E-Lock);
    raise (N-Go);
    wait (!N);
    cobegin
    if S-Go & !E & S-Req & S-Go & S-Req then lower (NS-Lock) endif
    lower (N-Go) ! lower (N-Req)
    end;
    while E-Req do
      if S-Go & !E & S-Req then lower (NS-Lock) endif
    endwhile
  end; endloop

loop
  if S-Req then
    if !N-Lock & !N-Req then raise (NS-Lock) else delay 1 endif;
    wait (!E-Lock);
    raise (S-Go);
    wait (!S);
    cobegin
    if N-Go & !N-Req then lower (NS-Lock) endif
    lower (S-Go) ! lower (S-Req)
    end;
    while E-Req do
      if N-Go & !N-Req & N-Req then lower (NS-Lock) endif
    endwhile
  end; endloop

loop
  if E-Req then
    wait (!E-Lock);
    cobegin raise (E-Lock) ! raise (E-Go) endif;
    wait (!E);
    cobegin lower (E-Lock) ! lower (E-Go) ! lower (E-Req) endif
  end; endloop
endproc

Figure 6.3: The Corrected Traffic Controller Program

7. Conclusion

Although finite state systems occur in a variety of contexts and correctness is frequently an issue, we believe our verification technique may prove to be most useful in analyzing the correctness of sequential circuits. We believe, in fact, that this approach is already practical for small- and medium-size circuits. The example in section 6 shows how the model checker can be used in conjunction with a high level language for describing PLA’s and ROM’s, and in [10] we show how state-transition graphs can be extracted for analysis from a switch level circuit specification.
However, more research is needed to make our method practical for large circuits. Circuit designers cope with the complexity of large circuits by designing them hierarchically. It seems reasonable that large circuits could be verified hierarchically by verifying small subcircuits in detail, then using simplified models of them as components in larger circuits. This process can be automated to some extent. If one uses a subset of CTL, small circuits can be simplified by "hiding" or restricting the visibility of some of their internal nodes (more precisely by making it illegal to use the nodes in CTL formulas and merging groups of states that become indistinguishable into single states). We refer the reader to [10] for a more detailed discussion of how model checking might be made hierarchical.

Perhaps the most difficult remaining problem involves determining how to handle replication of components in circuits. Consider, for example, the circuit for a queue. In this case the circuit actually represents a family of circuits in which each member has a different number of cells for storing queue elements. There are many other families of circuits designed in this way—systolic arrays in particular. It seems possible to verify entire families of such circuits at one time by using some type of induction rule in addition to the model checker. This, however, is a topic for future research.
References


