CMU-CS-85-100

Automatic Verification of Sequential Circuits using Temporal Logic

M. Browne, E. Clarke, D. Dill, and B. Mishra

Department of Computer Science,
Carnegie-Mellon University,
Pittsburgh, Pennsylvania 15213.

December, 1984

This research was supported by NSF. Grant Number MCS-82-16706.

ABSTRACT. *Verifying the correctness of sequential circuits has been an important problem for a long time. But lack of any formal and efficient method of verification has prevented the creation of practical design aids for this purpose. 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. In this paper we describe an automatic verification system for sequential circuits in which specifications are expressed in a propositional temporal logic. In contrast to most other mechanical verification systems, our system does not require any user assistance and is quite fast—experimental results show that state machines with several hundred states can be checked for correctness in a matter of seconds!*

The verification system uses a simple and efficient algorithm, called a Model Checker. *The algorithm works in two steps: in the first step, it builds a labeled state-transition graph; and in the second step, it determines the truth of a temporal formula with respect to the state-transition graph. We discuss two different techniques that we have implemented for automatically generating the state-transition graphs: The first involves extracting the state graph directly from the circuit by simulation. The second obtains the state graph by compilation from an HDL specification of the original circuit. Although these approaches are quite different, we believe that there are situations in which each is useful.*

1. Introduction

Verifying the correctness of sequential circuits has been an important problem for a long time. But lack of any formal and efficient method of verification has prevented the creation of practical design aids for this purpose. 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. In this paper we describe an automatic verification system for sequential circuits in which specifications are expressed in a propositional temporal logic. In contrast to most other mechanical verification systems, our system is fully automatic and does not require user assistance in the construction of proofs. Also, it is quite fast; experimental results show that state machines with several hundred states can be checked for correctness in a matter of seconds!

Propositional logic has long been accepted as an appropriate formalism for describing and reasoning about combinational circuits. We believe that temporal logic may be equally useful for sequential circuits. Bochmann [3] was probably the first to use temporal logic to describe circuits. He verified an implementation of a self-timed arbiter using linear temporal logic and what he called "reachability analysis." Malachi and Owicki [11] identified additional temporal operators required to express interesting properties of circuits and also gave specifications of a large class of modules used in self-timed systems. Although these researchers contributed significantly toward developing an adequate notation for expressing the correctness of circuits, the problem of mechanically verifying a circuit using efficient algorithms still remained unsolved.

We show how a simple and efficient algorithm, called a *model checker*, can be used to verify various temporal properties of a sequential circuit. Roughly speaking our method works by first building a labeled state-transition graph for the circuit. This graph can be viewed as a finite *Kripke structure* or *model*. By using the model checker we can determine the truth of a temporal formula relative to the state graph. Our algorithm has time complexity linear in both the size of the specification and the size of the state-transition graph. Moreover, if the formula is not true, the model checker will provide a counterexample if possible.

Thus, if we have correctly translated the circuit specification into a state-transition graph, we will know that a formula determined to be true by the model checker must also hold true for the corresponding circuit. We discuss two different techniques that we have implemented for automatically generating such graphs: The first involves extracting the state graph directly from the circuit by simulation. The second obtains the state graph by compilation from an HDL specification of the original circuit. Although these approaches are quite different, we believe that there are situations in which each is useful.

In the first approach a mixed gate- and switch- level circuit simulator is used to extract a state graph from a structural description of the sequential circuit. Usually, circuits are designed under the assumptions that some input sequences and combinations will not occur. The program exploits this 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 simulator uses a

unit-delay timing model in which the switching delays of all transistors and gates are assumed to be equal.

The second approach involves extracting the Kripke structure from a high-level state machine description language with a Pascal-like syntax (called SML). Since programs in the SML language must ultimately compile into circuits, the major data type is *boolean*. Furthermore, boolean variables may declared *active high* or *active low*, and use of mixed logic is encouraged. Programs are composed using the standard control structures **if**, **while**, and **loop/exit**. A **cobegin** is provided for simultaneous execution of statements, and there is a simple macro mechanism. 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.

The paper is organized as follows: Section 2 briefly describes the CTL specification language and how the model checker works. Section 3 discusses the automatic procedure that we have implemented for extracting a CTL model directly from a circuit and section 4 illustrates its use in verifying an asynchronous circuit from Seitz's chapter in Mead and Conway [12]. In section 5 we outline the alternative approach of extracting a CTL model 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 6 with a discussion of directions for future research including the possibility of making our approach hierarchical.

2. CTL and EMC

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL (Computation Tree Logic). This logic is essentially the same as that described in [1], [6] and [9]. The syntax for CTL is as follows: Let P be the set of all the atomic propositions in the language, $\mathcal{L}$. Then

1. Every atomic proposition P in P is a formula in CTL.
2. If f_1 and f_2 are CTL formulæ, then so are $\neg f_1$, $f_1 \wedge f_2$, $\mathbf{AX}\ f_1$, $\mathbf{EX}\ f_1$, $\mathbf{A}[f_1\ \mathbf{U}\ f_2]$ and $\mathbf{E}[f_1\ \mathbf{U}\ f_2]$.

In this logic the propositional connectives $\neg$ and $\wedge$ have their usual meanings of negation and conjunction. The temporal operator **X** is the nexttime operator. Hence the intuitive meaning of $\mathbf{AX}\ f_1$ ($\mathbf{EX}\ f_1$) is that f_1 holds in every (in some) immediate successor state of the current state. The temporal operator **U** is the *strong until* operator. The intuitive meaning of $\mathbf{A}[f_1\ \mathbf{U}\ f_2]$ ($\mathbf{E}[f_1\ \mathbf{U}\ f_2]$) is 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 also use the following syntactic abbreviations:

- $f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$, $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$, and $f_1 \leftrightarrow f_2 \equiv (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$

- $\mathbf{AF}\ f_1 \equiv \mathbf{A}[\mathbf{true}\ \mathbf{U}\ f_1]$ which means for every path, there exists a state on the path at which f_1 holds.
- $\mathbf{EF}\ f_1 \equiv \mathbf{E}[\mathbf{true}\ \mathbf{U}\ f_1]$ which means for some path, there exists a state on the path at which f_1 holds.
- $\mathbf{AG}\ f_1 \equiv \neg\,\mathbf{EF}\ \neg f_1$ which means for every path, at every node on the path f_1 holds.
- $\mathbf{EG}\ f_1 \equiv \neg\,\mathbf{AF}\ \neg f_1$ which means for some path, at every node on the path f_1 holds.

We also define the *weak until* operator, **u**; the 'universal form' of it can be defined as the following syntactic abbreviation: $\mathbf{A}[f_1\ \mathbf{u}\ f_2] \equiv \neg\,\mathbf{E}[\neg f_2\ \mathbf{U}\ (\neg f_1 \wedge \neg f_2)]$ which means that for every computation path, f_1 is true in all states preceding the (first) state in which f_2 is true. It is different from the strong until operator in the sense that it is does not imply eventual occurrence of its second argument.

The semantics of a CTL formula is defined with respect to a labelled state-transition graph. A CTL structure is a triple $M = (S, R, \Pi)$ where

1. *S is a finite set of states.*
2. *R is a total binary relation on S ($R \subseteq S \times S$) and denotes the possible transitions between states.*
3. *Π is an assignment of atomic proposition to states, i.e. $\Pi : S \rightarrow 2^P$.*

A *path* is an infinite sequence of states $(s_0, s_1, s_2, \ldots)$ such that $\forall_i[\langle s_i, s_{i+1}\rangle \in R]$. For any structure $M = (S, R, \Pi)$ and state $s_0 \in S$, there is an *infinite computation tree* with root labelled s_0 such that $s \rightarrow t$ is an arc in the tree *iff* $\langle s, t\rangle \in R$.

The truth in a structure is expressed by $M, s_0 \models f$, meaning that the temporal formula f is satisfied in the structure M at state s_0. The semantics of temporal formulæ is defined inductively as follows:

- $s_0 \models P$ *iff* $P \in \Pi(s_0)$.
- $s_0 \models \neg f$ *iff* $s_0 \not\models f$.
- $s_0 \models f_1 \wedge f_2$ *iff* $s_0 \models f_1$ and $s_0 \models f_2$.
- $s_0 \models \mathbf{AX}\ f_1$ *iff* for all states t such that $\langle s_0, t\rangle \in R, t \models f_1$.
- $s_0 \models \mathbf{EX}\ f_1$ *iff* for some state t such that $\langle s_0, t\rangle \in R, t \models f_1$.
- $s_0 \models \mathbf{A}[f_1\ \mathbf{U}\ f_2]$ *iff* for all paths $(s_0, s_1, s_2, \ldots), \exists_{i \geq 0}[s_i \models f_2 \wedge \forall_{0 \leq j < i}[s_j \models f_1]]$.
- $s_0 \models \mathbf{E}[f_1\ \mathbf{U}\ f_2]$ *iff* for some path $(s_0, s_1, s_2, \ldots), \exists_{i \geq 0}[s_i \models f_2 \wedge \forall_{0 \leq j < i}[s_j \models f_1]]$.

There is a program called EMC ("Extended Model Checker") that verifies the truth of a formula in a model using these definitions. It uses efficient graph-traversal algorithms to check a formula in time linear in the size of the graph and in the length of the formula. (See [6] for details.)

There are two additional features of the model checker that turn out to be particularly useful in practice. The first extension is the addition of *fairness constraints*. Occasionally, we are only interested in the correctness of fair execution sequences. For example, we may wish to consider

only execution sequences in which some process that is continuously enabled will eventually fire. This type of property cannot be expressed directly in CTL. 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. The path quantifiers in CTL formulas are now restricted to fair paths. Examples of fairness constraints can be found in sections 4 and 5. In [6] we show that handling fairness in this manner does not change the linear time complexity of the model checker.

The second feature is a counterexample facility. 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 $\neg f$ holds. This feature is quite useful for debugging. EMC is written in C and runs on a VAX 11/780 under Unix.

3. Extracting State Graphs from Circuits

Perhaps the most common approach to understanding a circuit is to trace its operation by propagating sample values through one level of gates completely before proceeding with the next. This is an application of a *unit delay* timing model: One assumes that the delay between input signals and the corresponding output is exactly the same for all of the gates in the circuit. The unit delay assumption is frequently used for simulation at the gate and switch level.

We use the unit-delay assumption when *verifying* sequential circuits. In essence, we have automated the informal process of checking the circuit operation for all possible inputs. The unit-delay assumption may not catch all errors in asynchronous circuits. A circuit may malfunction only in the presence of unequal gate delays, in which case some other method must be used to detect the possible error. However, we believe that verification under the unit-delay assumption is a good way to debug many types of asynchronous circuits—perhaps as an initial step in a more thorough (and expensive) verification process.

We describe below a program that converts a mixed gate- and switch-level description of a circuit into a state graph (called the *circuit processor* from now on). The resulting state graph and verification conditions (written in CTL) can then be fed to the model checker to do the actual checking.

The input to the circuit processor is a structural circuit description. It consists of a set of node and component declarations. A node can be declared to be an *input node* in which case it is assumed to be driven by an off-chip signal; otherwise, it is assumed to be in internal node serving as a connection point. Internal nodes also have the ability to store signals capacitively if they are not connected (directly or indirectly) to a power source. A component can be regarded as a box with a set of "formal nodes," similar to formal parameters in procedures in conventional programming languages. A component declaration consists of a component type and an association of its formal

nodes with the actual nodes in the circuit. Components can be resistors, transistors, or boolean gates.

Once the circuit description has been read, the circuit processor builds a state graph. The heart of the program is a mixed gate- and switch-level simulator. The states of the output state graph are characterized by the signals at the circuit nodes: There is at most one state corresponding to any set of node values and each state is labeled with the signals that have the value 1 in that state.

The construction of the state graph starts with a user-specified initial state and uses the simulator to find the successors to every new state it generates. The circuit processor also decides what values to use for the input nodes when finding the successors. A state can have more than one successor if there are several possible input values.

The simulation algorithm is the same as the one used in MOSSIM II, a widely used switch-level simulator. The algorithm and MOSSIM II are thoroughly explained elsewhere, so we summarize it only briefly here. (See [5].) The algorithm uses three "logical" values: 0, 1 and X (meaning "unknown"). [1]

Basically, the simulation performs a set of steps, each step simulating one unit delay. There are logical values assigned to the circuit nodes on entry to each step, either from the results of the previous step or from user-specified initial conditions. A step consists of two phases. First, the logical node values are used to determine whether the transistors are "on," "off," or "unknown." The transistors are frozen in this state giving a resistor circuit. In the second phase the node values and the resistor circuit are used to find new node values using an approximate model. This gives the logical node values for the next step.

We have augmented the algorithm to allow simulation of arbitrary boolean gates. During the first phase of a unit step a boolean result is computed from the logical values of the gate's input nodes. If the gate has a boolean output of 1, the simulator adds a pullup resistor to the gate output node in the resistor circuit used in phase 2. If the boolean result is 0, the simulator adds both a pullup and an appropriate pulldown resistor to the output node.

Since circuits are only expected to work when they are properly used, the program need not consider all inputs at all states. Instead, the user specifies a set of conditions under which the inputs can change. The conditions are propositional formulas on the node values. The circuit processor determines whether a state satisfies the change condition for each input signal, and uses this information to decide whether that signal can change during the computation of the successor states. Each signal that can change has two possible values: the current value and its logical complement. Any combination of these values is a possible input. It is possible for the inputs to remain unchanged in every state.

---

[1] This presents no problem for the model checker. We have extended it to accept a state graph in which the states have two sets of labels: one of nodes with 1 values and one of nodes with 0 values. A node with an X value appears in neither set. A state graph of this form represents a family of state graphs of the previous type; each member of the family is the result of substituting 0 or 1 for a node that has the value X in the original graph. A formula is satisfied by a state iff it is satisfied by the corresponding state in every element of this family of graphs.

```
-- 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 state for the node values in the table
    then
        return the state;
    else
        Create a new state;
        Label the state with nodes that have 1 values;
        Store the state and node values together in the 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 the previous line to the successors of the
            current state;
        end;
    end;
end.
```

Figure 1: State Machine Construction Algorithm.

It is sometimes useful for an input condition to be able to test whether a state is "stable." (In the unit delay model, a stable state is a state that is a successor of itself. The circuit can stay in such a state for an arbitrarily long time.) Usually, for example, a clock signal in a synchronous circuit should not change until the circuit is stable. We allow the atomic proposition *stable* to be used in input change conditions. The preprocessor labels a state with *stable* if that state is a successor of itself. This occurs when the circuit has settled; it will stay in a stable state until the inputs change.

For example, the queue element we describe and verify below has an input signal named *Init* to initialize it. The circuit should be verified under conditions in which *Init* is raised, stays high until the circuit has stabilized (no other inputs may change during this time), then goes low and stays low. We give *Init* the value 1 at the beginning of the state graph construction (the program asks for this information). In the circuit description, the input change condition for *Init* is given by $Init \wedge stable$. This requires *Init* to stay high until the circuit stabilizes. It can then continue to be high, or go low. Once it goes low, it cannot go high again because the change condition says that it can only change when it is high.

A more detailed description of the program appears in Figure 1.

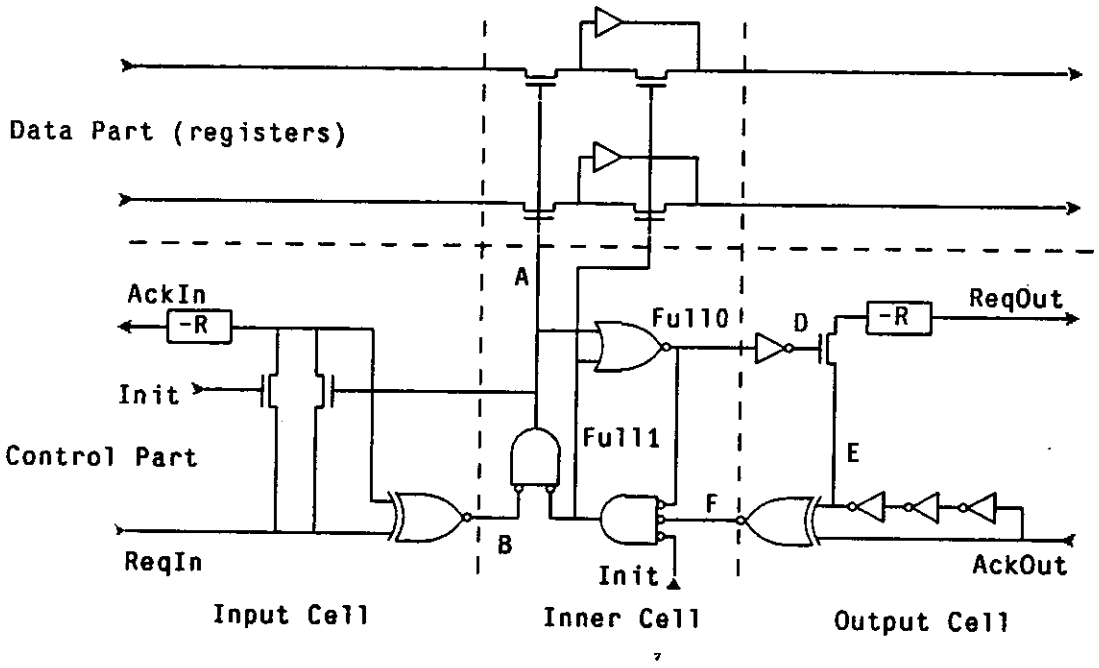

Figure 2: Queue (FIFO) Element.

4. Example: A Self-Timed Queue Element

We apply this technique to a self-timed queue element. The circuit originally appeared in an article by Seitz on self-timed systems [14]. This circuit has practical importance because it can be used to connect pipelined computational units with variable processing time, maximizing the utilization of the connected units. The use of asynchronous design results in a very fast and small implementation of the queue. A diagram of the circuit is shown in Figure 2. The queue consists of a control part and a data part. The data part is simply a shift register which has as many bits of parallel data as necessary. The control part has three major components: an *input cell*, an *output cell*, and any number of *inner cells*. The input and output cells convert two-cycle signals from the external world into four-cycle signals for the inner cells. The inner cells keep track of which cells of the shift register have data in them and handle the control signals to shift data through the register.

There is one unconventional component in the circuit that should be explained: the "negative resister" (labeled "$-R$" in the diagram). This circuit is a chain of two inverters with a (relatively) high-value resistor feeding back from the output of the second inverter to the input of the first. This circuit stores its most recent input signal, and after two gate delays supplies the same signal on the output.

The queue is a speed-independent element: It assumes no real-time restrictions on the behavior of the circuits it is connected to. However, the internal design of the queue is *not* speed-independent. It uses a more liberal assumption that no series of 3 gates is faster than any other series of 2 gates. This is called the "3/2 rule".

We have applied our verification technique to the circuit in the case where there is a single inner cell. The unit-delay assumption is a refinement of the 3/2 rule. Any circuit satisfying the unit-delay rule certainly satisfies the 3/2 rule, but not the converse. If our verification finds a problem, then it is definitely a violation of the circuit design rules. On the other hand, a successful verification increases confidence in the circuit design but does not guarantee that the circuit is correct.

4.1. Temporal Logic Specification of the Queue Element

In this subsection we give a variety of correctness conditions in CTL for the queue element. This is not a complete specification—just a sample of some interesting properties. We categorize the conditions as requiring *safety* or *liveness* properties. Informally, safety properties say that the circuit does not do anything bad, while liveness properties say that it does do something good.

First, we specify the correct behavior of the two-cycle interfaces with the external world. The following formulæ apply to both the input and output cells. *ReqIn*, *ReqOut*, *AckIn*, and *AckOut* must be substituted for *req* and *ack*, as appropriate.

$$\mathbf{AG}\,(\neg req \rightarrow \mathbf{A}[\neg req\ \mathbf{u}\ \neg ack]) \qquad \mathbf{AG}\,(req \rightarrow \mathbf{A}[req\ \mathbf{u}\ ack])$$
$$\mathbf{AG}\,(\neg ack \rightarrow \mathbf{A}[\neg ack\ \mathbf{u}\ req]) \qquad \mathbf{AG}\,(ack \rightarrow \mathbf{A}[ack\ \mathbf{u}\ \neg req])$$

The first condition requires that if the *req* signal is low it must stay low until *ack* goes low; if *ack* is high because a previous request has not been acknowledged *req* is not allowed to change. The second formula gives the corresponding requirement when *req* is high. The third and fourth formulæ require that *ack* not change unless *req* has the opposite value.

These previous four formulæ are *safety* properties. For example, in the first formula it is not required that $\neg ack$ go low—only that *req* cannot go high before *ack* goes low. It is also reasonable to give some liveness conditions for the two-cycle interfaces:

$$\mathbf{AG}\,(req \wedge \neg ack \rightarrow \mathbf{AF}\ ack) \qquad \mathbf{AG}\,(\neg req \wedge ack \rightarrow \mathbf{AF}\ \neg ack)$$

These formulæ state that if every request must inevitably be acknowledged. We do *not* require *req* to change after *ack* takes on the same value.

There is a problem in verifying these last two formulæ. In a correct implementation of the queue element, if the register cell is already full and another input request arrives, the acknowledge for the new input must wait for the cell to become empty. This will only happen when the external circuit on the output side raises *AckOut* to indicate that it has read the contents of the register cell. We must require that this external circuit always respond to an output request in finite time.

The solution to this problem is to use the fairness constraint facility of the model checker. We can require that a pending output request be acknowledged infinitely often by the fairness constraint

$$ReqOut \leftrightarrow AckOut.$$

The model checker will then check the two conditions above only over the paths in which the external circuit always responds to output requests, which solves the problem.

There are also correctness conditions relating the input and output cells. We give a few sample formulæ. First, an obvious safety condition is that if there is nothing in the inner cell ($\neg Full1$) there will not be an output request until there is an input request,

$$\mathbf{AG}\,(\neg Full1 \rightarrow \mathbf{A}[(ReqOut \leftrightarrow AckOut)\ \mathbf{u}\ \neg(ReqIn \leftrightarrow AckIn)])).$$

There are also a number of interesting liveness conditions. For example, if there is an unacknowledged input request and if the inner cell is empty, then the signal to load the shift register cell, A, will inevitably be raised,

$$\mathbf{AG}\,(\neg(ReqIn \leftrightarrow AckIn) \wedge \neg Full1 \rightarrow \mathbf{AF}\ A).$$

If the inner cell is full, there should always be an output request to make the data available,

$$\mathbf{AG}\,(\neg Full1 \rightarrow \mathbf{AX}\,(Full1 \rightarrow \mathbf{AF}\ \neg(ReqOut \leftrightarrow AckOut))).$$

This formula is somewhat subtle. Essentially, the nexttime operator is used to check the first state after a rising edge of *Full1*. After this edge there must inevitably be an output request.

If the queue element is full and an output request is made and then acknowledged, the element should eventually become empty. This is specified by

$$\mathbf{AG}\,(Full1 \wedge \neg(ReqOut \leftrightarrow Ackout) \rightarrow \mathbf{AX}\,((ReqOut \leftrightarrow AckOut) \rightarrow \mathbf{AF}\ \neg Full1)).$$

4.2. Verifying the Circuit

When our program is used to build a state graph from a circuit description, the resulting state graph has 152 states. The input signal *Init* is set to 1 and the signals *ReqIn* and *AckOut* are set to 0 for the starting conditions (all internal signals are automatically initialized to a value that means "unknown"). We specify input change conditions requiring that the *Init* signal stay high until the circuit stabilizes, and that none of the other inputs change until *Init* goes low. Thus, the resulting state graph has a sequence of states at the beginning representing the initialization of the circuit, after which the *Init* signal stays low and the other signals are free to change. There are also input change conditions to require *ReqIn* and *AckOut* to conform to the two-cycle signalling protocol.

The actual verification conditions are modified to account for the *Init* signal: Every condition originally of the form $\mathbf{AG}\,(x)$ is transformed to $\mathbf{AG}\,(\neg Init \rightarrow x)$, so uninitialized states are not considered in checking the formulæ.

All the conditions check, except for the safety condition

$$\mathbf{AG}\,(\neg Full1 \rightarrow \mathbf{A}[(ReqOut \leftrightarrow AckOut)\ \mathbf{u}\ \neg(ReqIn \leftrightarrow AckIn)]).$$

The model checker provides a counter-example path in which there are *two* output requests in response to a single input request. This occurs because of a timing error in the circuit. The transistor in the output cell connected to *ReqOut* is *on* (because $B = 1$) when *AckOut* goes high in response to *ReqOut*. The data path that eventually causes B to go low is four gates long, so B is still high when the change in *AckOut* propagates through the three inverters to E. This causes *ReqOut* to go low, creating a spurious request on the output.

It is not clear whether this phenomenon would cause a real circuit to fail—that depends on how accurately the 3/2 design rules model real circuits. However, the circuit definitely has a bug under those rules. This bug can be fixed by adding two more inverters between *AckOut* and E. All of the above conditions check out for the corrected circuit; however, we still cannot be sure that the resulting circuit is bug-free because the specification is incomplete and because we have not checked it under the less forgiving 3/2 timing model.

5. Verifying High Level Descriptions of Circuits

In practice, many circuits are designed as finite state machines before they are implemented in hardware. For circuits designed in this manner, exhaustive simulation that constructs a finite state machine, as in the previous section, is unnecessary since the original finite state machine is already available. Therefore, we can verify the design before it is implemented in hardware. If a VLSI design tool that correctly implements finite state machines is used to layout the verified design, we can be sure that the resulting circuit is correct.

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.

5.1. 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 are 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 they will appear in a forthcoming paper [4]. A *program state* is an ordered pair, $\langle S, s \rangle$, consisting of a statement S and a function s that gives values to all of the identifiers. The semantics consist of a set of *rewrite rules* that describe how a program state can be transformed into 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:

$$\langle \mathbf{raise}(I); S, s \rangle \xrightarrow{1} \langle S, s' \rangle \quad \text{where } s' = s\,[I \mapsto \mathit{true}] \tag{1}$$

$$\frac{E = \mathit{false}}{\langle \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{endif}; S_2, s \rangle \xrightarrow{0} \langle S_2, s \rangle} \tag{2}$$

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 s so that $s'(I) = \mathit{true}$. 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 *false* to all booleans), we can repeatedly find successor states until we have built the entire finite state machine.

5.2. 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-Go*, *S-Go*, and *E-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.

$$\mathbf{AG}\ \neg(E\text{-}Go \wedge (N\text{-}Go \vee S\text{-}Go))$$

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:

$$\mathbf{AG}\ (\neg N\text{-}Go \wedge N \rightarrow \mathbf{AF}\ N\text{-}Go)$$
$$\mathbf{AG}\ (\neg S\text{-}Go \wedge S \rightarrow \mathbf{AF}\ S\text{-}Go)$$
$$\mathbf{AG}\ (\neg E\text{-}Go \wedge E \rightarrow \mathbf{AF}\ E\text{-}Go)$$

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.

$$\mathbf{EF}\ (N\text{-}Go \wedge S\text{-}Go)$$

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-Go* is true, and it stays there while *N* is true, the fairness constraint for cars from the north is $\neg(N\text{-}Go \wedge N)$. There are similar constraints for traffic from the south and east.

5.3. An implementation of the Traffic Controller in SML

One approach to this problem is to provide two locks: `NS-Lock`, which is true when north-south traffic is in the intersection, and `EW-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 3 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-Req`, `S-Req`, and `E-Req` are also declared to be internal. `N-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-Req` and `E-Req` are similar.

Lines 7–9: `Wait` is a macro definition that delays until its parameter becomes true.

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.

```
1       program intersect;
2
3       input N, S, E;
4       output N-Go, S-Go, E-Go;
5       internal NS-Lock, EW-Lock, N-Req, S-Req, E-Req;
6
7           procedure wait (expr)
8               while !(expr) do nop endwhile
9           endproc
10
11          cobegin
12              loop if !N-Req & N then raise (N-Req) endif endloop
13          ||
14              loop if !S-Req & S then raise (S-Req) endif endloop
15          ||
16              loop if !E-Req & E then raise (E-Req) endif endloop
17          ||
18              loop
19                  if N-Req then
20                      raise (NS-Lock);
21                      wait (!EW-Lock);
22                      raise (N-Go);
23                      wait (!N);
24                      cobegin
25                          if !S-Go & !S-Req | S-Go & !S then lower (NS-Lock) endif
26                      ||
27                          lower (N-Go) || lower (N-Req)
28                      end;
29                      wait (!E-Req)
30                  endif
31              endloop
32          ||
33              loop
34                  if S-Req then
35                      if !NS-Lock & !N-Req then raise (NS-Lock) else delay 1 endif;
36                      wait (!EW-Lock);
37                      raise (S-Go);
38                      wait (!S);
39                      cobegin
40                          if !N-Go & !N-Req then lower (NS-Lock) endif
41                      ||
42                          lower (S-Go) || lower (S-Req)
43                      end;
44                      wait (!E-Req)
45                  endif
46              endloop
47          ||
48              loop
49                  if E-Req then
50                      wait (!NS-Lock);
51                      cobegin raise (EW-Lock) || raise (E-Go) end;
52                      wait (!E);
53                      cobegin lower (EW-Lock) || lower (E-Go) || lower (E-Req) end
54                  endif
55              endloop
56          end
57      endprog
```

Figure 3: A First Attempt at Writing a Traffic Controller in SML.

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.

5.4. Verifying the Traffic Controller with the Model Checker

Figure 4 shows a transcript of the model checker running on the program in figure 3. 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 3 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 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 5. 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.

```
% /bin/time emc -c inter1.emc
                CTL MODEL CHECKER (C version 2.5)

Taking input from inter1.emc...
Fairness constraint: ~(N-Go & N).
Fairness constraint: ~(S-Go & S).
Fairness constraint: ~(E-Go & E).
Fairness constraint: .

time:  (1284 141)

|= EF (N-Go & S-Go).
The equation is TRUE.


time:  (1305 149)

|= AG ~(E-Go & (N-Go | S-Go)).
The equation is TRUE.


time:  (1326 158)

|= AG (N & ~N-Go -> AF N-Go).
The equation is FALSE.

EF ~(N & ~N-Go -> AF N-Go)
        is true in state 1 because of the path:
State 1:  XCMP1 E S N
State 16:  E-Req S-Req N-Req XCMP2
State 104:  E-Req S-Req N-Req E-Go EW-Lock NS-Lock XCMP67
State 484:  E S-Req N-Req NS-Lock XCMP32
State 390:  S E-Req S-Req N-Req NS-Lock S-Go N-Go XCMP30
State 417:  E S N E-Req S-Req NS-Lock S-Go XCMP19

N & ~N-Go -> AF N-Go
        is false in state 417 if:
        1) ~(N & ~N-Go)
                is false in state 417, AND
        2) AF N-Go
                is false in state 417.

~(N & ~N-Go)
        is false in state 417 because the following propositions are true:
N ~N-Go

AF N-Go
        is false in state 417 because
EG ~N-Go
        is true in state 417.
An example of such a path is:
State 417:  E S N E-Req S-Req NS-Lock S-Go XCMP19
State 432:  E-Req S-Req N-Req NS-Lock S-Go XCMP17
State 523:  E N E-Req N-Req NS-Lock XCMP69
State 523:  E N E-Req N-Req NS-Lock XCMP69
        ...


time:  (1428 184)

|= .
End of Session.
     1:59.0 real        23.8 user        3.2 sys
%
```

Figure 4: Verifying the First Traffic Controller Program.

6. Conclusion

The approaches presented here are practical for small- and medium-size sequential circuits. Verification is usually viewed as a way to guarantee correctness, and these techniques are no

```
1       program intersect;
2
3       input N, S, E;
4       output N-Go, S-Go, E-Go;
5       internal NS-Lock, E-Lock, N-Req, S-Req, E-Req;
6
7           procedure wait (expr)
8               while !(expr) do nop endwhile
9           endproc
10
11          cobegin
12              loop if !N-Req & N then raise (N-Req) endif endloop
13          ||
14              loop if !S-Req & S then raise (S-Req) endif endloop
15          ||
16              loop if !E-Req & E then raise (E-Req) endif endloop
17          ||
18              loop
19                  if N-Req then
20                      raise (NS-Lock);
21                      wait (!E-Lock);
22                      raise (N-Go);
23                      wait (!N);
24                      cobegin
25                          if S-Go & !S | !S-Go & !S-Req then lower (NS-Lock) endif
26                      ||
27                          lower (N-Go) || lower (N-Req)
28                      end;
29                      while E-Req do
30                          if S-Go & !S & N-Req then lower (NS-Lock) endif
31                      endwhile
32                  endif
33              endloop
34          ||
35              loop
36                  if S-Req then
37                      if !NS-Lock & !N-Req then raise (NS-Lock) else delay 1 endif;
38                      wait (!E-Lock);
39                      raise (S-Go);
40                      wait (!S);
41                      cobegin
42                          if !N-Go & !N-Req then lower (NS-Lock) endif
43                      ||
44                          lower (S-Go) || lower (S-Req)
45                      end;
46                      while E-Req do
47                          if N-Go & !N & S-Req then lower (NS-Lock) endif
48                      endwhile
49                  endif
50              endloop
51          ||
52              loop
53                  if E-Req then
54                      wait (!NS-Lock);
55                      cobegin raise (E-Lock) || raise (E-Go) end;
56                      wait (!E);
57                      cobegin lower (E-Lock) || lower (E-Go) || lower (E-Req) end
58                  endif
59              endloop
60          end
61      endprog
```

Figure 5: The Corrected Traffic Controller Program.

exception. However, we believe that these methods hold even more promise as debugging aids. Tools like those described in this paper could expedite the design process by localizing bugs quickly. They could also allow designers to improve designs more aggressively, freeing them from the natural reluctance to modify a design that is already known to work.

Further research is needed in a number of areas. Timing is an important issue when verifying asynchronous sequential circuits. The unit-delay model used in sections 3 and 4 is easy to im-

```
% /bin/time emc -c inter2.emc
                CTL MODEL CHECKER (C version 2.8)

Taking input from inter2.emc...
Fairness constraint: ~(N-Go & N).
Fairness constraint: ~(S-Go & S).
Fairness constraint: ~(E-Go & E).
Fairness constraint: .

time:  (1371 365)

|= AG ~(E-Go & (N-Go | S-Go)).
The equation is TRUE.


time:  (1391 371)

|= AG (N & ~N-Go -> AF N-Go).
The equation is TRUE.


time:  (1465 377)

|= AG (S & ~S-Go -> AF S-Go).
The equation is TRUE.


time:  (1543 383)

|= AG (E & ~E-Go -> AF E-Go).
The equation is TRUE.


time:  (1609 391)

|= .
End of Session.
     4:51.0 real        26.8 user         6.7 sys
%
```

Figure 6: Verifying the Corrected Traffic Controller Program.

plement, but unrealistic. A more commonly used model in asynchronous circuit design assumes arbitrary delays in wires and/or gates. We have a technique for verifying circuits under an arbitrary gate delay model, which we have successful applied to an asynchronous arbiter [8]. There are a variety of timing assumptions that are less conservative than arbitrary delay models, but more realistic than the unit-delay assumption. Obviously, the 3/2 model used in the design of the queue element example is one of these. Another assumes minimum and maximum delays for the circuit components. It would be useful to be able to verify circuits under these assumptions.

It is probably not practical to use these methods on large circuits, because of the corresponding size of the state graphs. Circuit designers cope with the complexity of large circuits by designing them hierarchically. It seems reasonable that the same 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" some of their internal nodes (more precisely, making it illegal to use them in CTL formulæ) and merging groups of states that become indistinguishable into single states (this is called *restriction*) [7].

We verified the self-timed queue element in the specific case in which there was only one inner cell. In fact, there is a family of queues, each member having a different number of repeated inner cells. There are many families of circuits designed in this way, for example, systolic arrays in which the number of cells is a parameter. It would be useful to be able to verify entire families of circuits at one time, using a more general technique than the ones in this paper. We conjecture that inductive techniques could be applied to this problem.

REFERENCES

[1] M.BEN-ARI, Z.MANNA AND A.PNUELI, "The Logic of Nexttime," *Eighth ACM Symposium on Principle of Programming Languages*, Williamsburg, VA, January 1981.

[2] G.BERRY AND L.COSSERAT, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," Ecole Nationale Supérieune des Mines de Paris (ENSMP), Cetnre de Mathématiques Appliquées, Sophia-Antipolis, 06565 Valbonne, France, 1984.

[3] G.V.BOCHMANN, "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers*, Vol C-31, No. 3, March 1982.

[4] M.C.BROWNE AND E.M.CLARKE, *Unpublished Manuscript*, December 1984.

[5] R.E.BRYANT, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, Vol C-33, No. 2, February 1984.

[6] E.M.CLARKE, E.A.EMERSON AND A.P.SISTLA, "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications: A Practical Approach," *Tenth ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983.

[7] E.CLARKE AND B.MISHRA, "Automatic Verification of Asynchronous Circuits," in *Proceedings of C-M.U. Workshop on Logics of Programs* (ed. E. Clarke and D.Kozen), Pittsburgh, PA, 1983 (Springer Lecture Notes in Computer Science).

[8] D.DILL, *Unpublished Manuscript*, December 1984.

[9] E.A.EMERSON AND E.M.CLARKE, "Characterizing Properties of Parallel Programs as Fixpoints," *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science No. 85, Springer Verlag, 1981.

[10] J.HALPERN, Z.MANNA AND B.MOSZKOWSKI, *A Hardware Semantics based on Temporal Intervals*, Report No. STAN-CS-83-963, Department of Computer Science, Stanford University, Stanford University, Stanford, CA 94305, March 1983.

[11] Y.MALACHI AND S.S.OWICKI, "Temporal Specifications of Self-Timed Systems," in *VLSI Systems and Computations* (Ed. H.T.Kung, Bob Sproull, and G.Steele), Computer Science Press, 1981.

[12] C.A.MEAD AND L.A.CONWAY, *Introduction to VLSI Systems*, Reading, MA, Addison-Wesley, 1980.

[13] R.MILNER, *A Calculus of Communicating Systems*, University of Edinburgh, June 1980.

[14] C.SEITZ, "System Timing," in *Inroduction to VLSI Systems* (C.Mead and L.Conway), Reading, MA, Addison-Wesley, 1980.