Automatic Circuit Verification Using Temporal Logic: Two New Examples

Michael C. Browne
Edmund M. Clarke
David L. Dill
Carnegie-Mellon University
Pittsburgh Pennsylvania 15213

1. Introduction

Temporal logic is a formal system for reasoning about the occurrence of events in time. It belongs to the class of modal logics, originally developed by philosophers in an attempt to classify logical propositions according to their "possibility" or "impossibility". In the case of temporal logic special operators are introduced that describe how the truth values of assertions vary with time. A typical operator is G f which is true now if f is true at all future moments (f is Globally true). As an example of how temporal logic might be used, consider the assertion that two events S_1 and S_2 do not occur simultaneously. This is naturally expressed by the formula $G(\neg S_1 \vee \neg S_2)$.

Pnueli [14] was apparently the first to realize that temporal logic might be useful for reasoning about concurrent programs. Later, Bochmann [4] and Owicki and Malachi [13] showed how his ideas could be extended to handle sequential circuits. Although these researchers contributed significantly toward developing an adequate notation for specifying properties of circuits, the problem of showing that a circuit actually met its specifications still required a tedious hand-constructed proof.

In previous papers ([6], [7]) we have described an automatic, temporal logic based verifier for finite state concurrent systems. Our verifier uses a simple and efficient algorithm, called a *model checker*, to determine the truth of a temporal formula relative to a state transition graph. If the formula is not true, the model checker will provide a counterexample if possible. Unlike most verifiers, our algorithm has time complexity linear in both the size of the specification and the size of the state-transition graph. Experimental results show that the verifier can check temporal properties of state transition graphs at a rate of roughly a hundred states a second. Moreover, since the algorithm is linear in the size of the state graph we can expect the same rate for graphs with several thousand states that we get for graphs with several hundred states.

Sequential circuit verification is a natural application for the verifier. In ([1], [11], [8]) we have described how the verifier might be used to debug such circuits. The basic idea is to extract a state transition graph from some representation of the circuit and then use the model checker. In this paper we provide further evidence for the usefulness of our approach by describing enhancements to the basic verifier that automate the extraction of state transition graphs from circuits. We discuss two different techniques. The first approach involves extracting the state graph directly from a wire-list description of the circuit and is suitable for asynchronous circuits. The second obtains the state diagram 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.

2. The Logic and the Model Checker

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL (Computation Tree Logic). It is similar to those described in [3], [7], and [12]. The syntax that we use for CTL in this paper is given below. We assume the existence of an underlying set of atomic propositions AP for denoting events.

- Any atomic proposition $P \in AP$ is a CTL formula.
- If f and g are CTL formulas, then $\neg f$, $f \wedge g$, $\mathbf{G} f$, $\mathbf{F} f$, $f \mathbf{U} g$, and $f \mathbf{u} g$ are CTL formulas.

The symbols $\neg$ and $\wedge$ have their usual meanings. In addition to these formulas, we use the abbreviations $f \vee g$ and $f \rightarrow g$ for $\neg(\neg f \wedge \neg g)$ and $\neg(f \wedge \neg g)$, respectively. Intuitively, G f means that f is true in every state (Globally true), and F f means that f is eventually true somewhere along every sequence of states (true at some Future time). There are two versions of the *until* operator: the *strong until* "U" and the *weak until* "u". The *strong until* f U g requires that g must eventually be true and that f always hold until g is true. The *weak until* f u g also requires that f always be true until g becomes true; however, it is permissible for g never to be true if f remains true forever.

The semantics of CTL formulas is defined with respect to a labeled state transition graph. A CTL *structure* or

Automatic Circuit Verification Using Temporal Logic: Two New Examples

Michael C. Browne
Edmund M. Clarke
David L. Dill

Carnegie-Mellon University
Pittsburgh Pennsylvania 15213

1. Introduction

Temporal logic is a formal system for reasoning about the occurrence of events in time. It belongs to the class of modal logics, originally developed by philosophers in an attempt to classify logical propositions according to their "possibility" or "impossibility". In the case of temporal logic special operators are introduced that describe how the truth values of assertions vary with time. A typical operator is **G** f which is true now if f is true at all future moments (f is **G**lobally true). As an example of how temporal logic might be used, consider the assertion that two events S_1 and S_2 do not occur simultaneously. This is naturally expressed by the formula $\mathbf{G}(\neg S_1 \vee \neg S_2)$.

Pnueli [14] was apparently the first to realize that temporal logic might be useful for reasoning about concurrent programs. Later, Bochmann [4] and Owicki and Malachi [13] showed how his ideas could be extended to handle sequential circuits. Although these researchers contributed significantly toward developing an adequate notation for specifying properties of circuits, the problem of showing that a circuit actually met its specifications still required a tedious hand-constructed proof.

In previous papers ([6], [7]) we have described an automatic, temporal logic based verifier for finite state concurrent systems. Our verifier uses a simple and efficient algorithm, called a *model checker*, to determine the truth of a temporal formula relative to a state transition graph. If the formula is not true, the model checker will provide a counterexample if possible. Unlike most verifiers, our algorithm has time complexity linear in both the size of the specification and the size of the state-transition graph. Experimental results show that the verifier can check temporal properties of state transition graphs at a rate of roughly a hundred states a second. Moreover, since the algorithm is linear in the size of the state graph we can expect the same rate for graphs with several thousand states that we get for graphs with several hundred states.

Sequential circuit verification is a natural application for the verifier. In ([1], [11], [8]) we have described how the verifier might be used to debug such circuits. The basic idea is to extract a state transition graph from some representation of the circuit and then use the model checker. In this paper we provide further evidence for the usefulness of our approach by describing enhancements to the basic verifier that automate the extraction of state transition graphs from circuits. We discuss two different techniques. The first approach involves extracting the state graph directly from a wire-list description of the circuit and is suitable for asynchronous circuits. The second obtains the state diagram 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.

2. The Logic and the Model Checker

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL (Computation Tree Logic). It is similar to those described in [3], [7], and [12]. The syntax that we use for CTL in this paper is given below. We assume the existence of an underlying set of atomic propositions AP for denoting events.

- Any atomic proposition $P \in AP$ is a CTL formula.
- If f and g are CTL formulas, then $\neg f$, $f \wedge g$, $\mathbf{G}f$, $\mathbf{F}f$, $f\,\mathbf{U}\,g$, and $f\,\mathbf{u}\,g$ are CTL formulas.

The symbols $\neg$ and $\wedge$ have their usual meanings. In addition to these formulas, we use the abbreviations $f \vee g$ and $f \rightarrow g$ for $\neg(\neg f \wedge \neg g)$ and $\neg(f \wedge \neg g)$, respectively. Intuitively, **G** f means that f is true in every state (**G**lobally true), and **F** f means that f is eventually true somewhere along every sequence of states (true at some **F**uture time). There are two versions of the *until* operator: the *strong until* "U" and the *weak until* "u". The *strong until* $f\,\mathbf{U}\,g$ requires that g must eventually be true and that f always hold until g is true. The *weak until* $f\,\mathbf{u}\,g$ also requires that f always be true until g becomes true; however, it is permissible for g never to be true if f remains true forever.

The semantics of CTL formulas is defined with respect to a labeled state transition graph. A CTL *structure* or

model is a state transition graph consisting of:

- A finite set of states S;
- A function $L:S \rightarrow \mathcal{P}(AP)$, which labels each state with a set of atomic propositions;
- A transition relation $R \subseteq S \times S$;
- A start state, s_0.

We require that every state have at least one successor. A *path* from $s_i \in S$ is an infinite sequence of states that starts with s_i and in which R holds between every state and its successor.

Below we formally define what it means for a formula f to be *true in a state* s_i. This is written $s_i \models f$.

- $s_i \models P$ iff $P \in L(s_i)$.
- $s_i \models f \wedge g$ iff $s_i \models f$ and $s_i \models g$.
- $s_i \models \neg f$ iff it is not the case that $s_i \models f$.
- $s_i \models G f$ iff for every state s_j on every path from s_i, $s_j \models f$.
- $s_i \models F f$ iff there exists a state s_j on every path from s_i such that $s_j \models f$.
- $s_i \models f \, U \, g$ iff for every path from s_i there exists a state s_k such that $s_k \models g$ and for all states s_j preceding s_k on the path, $s_j \models f$.
- $s_i \models f \, u \, g$ iff for every path from s_i either
  1. there exists a state s_k such that $s_k \models g$ and for all states s_j preceding s_k on the path, $s_j \models f$, or
  2. for all states s_j on the path $s_j \models f$.

A formula is true in a state graph iff the formula is true in s_0. Figure 2-1 shows some examples of CTL formulas and state graphs in which they are true.

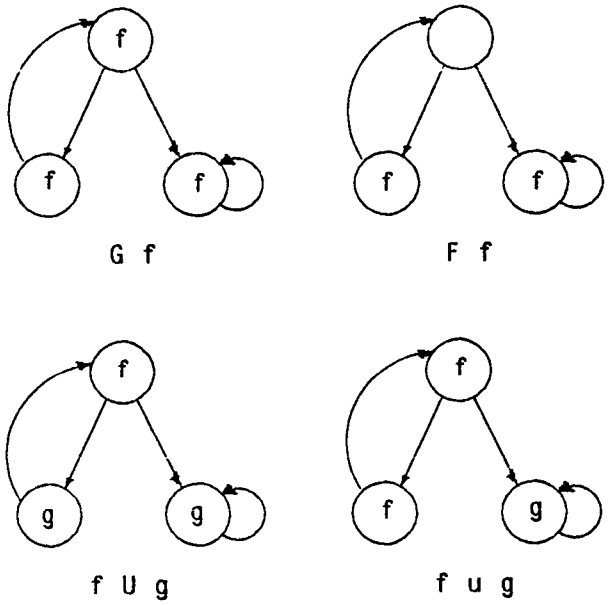

Figure 2-1: CTL Formulas and State Graphs in Which They Are True

There is a program called EMC ("Extended Model Checker") that verifies the truth of a formula in a model using these definitions. The algorithm processes a formula bottom up, checking the shortest subformulas before checking the subformulas that contain them. When it checks each subformula f, it labels every state in the graph with f if f is true in the state. Thus, when EMC is processing a formula it can treat the subformulas as atomic propositions. Processing any of the modal or propositional connectives requires at most a single depth-first traversal of the state graph, so the time to check a formula is proportional to the size of the state graph and size of the formula.

There is one very practical feature of the model checker that should be described: the 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 $G f$, 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. Verifying Asynchronous Circuits

The difficulty of designing correct asynchronous sequential circuits is well-known. Unlike synchronous circuits, in which concurrent activities proceed in lock-step, asynchronous circuits can have a large number of alternative execution sequences, each resulting from a different set of delays in the circuit elements. The circuit must ultimately produce correct results for any of these executions.

To be certain that an asynchronous design is correct, the designer must make delay assumptions that are guaranteed to hold for all instances of the circuit. One common delay model assumes that there is an arbitrary finite delay on every gate output. We term designs relying on this assumption *speed-independent.* This is a very conservative model, in the sense that any circuit that appears to be correct will work under a wide variety of realizations. However, it is often quite difficult to design circuits that are truly speed-independent. When it is possible, the resulting circuits are often unnecessarily complex and expensive, since "any reasonable implementation" would satisfy stronger delay assumptions.

We believe that the design of correct and practical asynchronous circuits could be greatly facilitated by a verification tool to automate the tedious analysis of circuit executions under varying delays. This tool should be able to use the designer's assumptions about circuit delays to avoid considering unreasonable circuit executions.

In the remainder of this section we describe in more detail the speed-independent circuit delay model and the algorithm for extracting a state graph from a circuit. We then discuss the delay assumptions and how they are incorporated into the state-graph construction algorithm. Finally, as an example, we describe, specify, and verify a "real" asynchronous circuit: a patented queue element.

3.1. Speed-Independent Circuits

The structural description of a circuit consists of a set of *elements*, which have *inputs* and *outputs*, and *nodes*, which are used to interconnect element inputs and outputs. Every element input and output must be connected to a node. Every node must be connected to *exactly one* element output and can be connected to any number of inputs.

Elements represent circuit components, which compute values from their inputs and, after some delay, supply these values on their outputs. Circuit elements are described using flow tables. Each element has a set of possible internal states. There is a *transition function* which gives a set of successor states, given a current state and values for the inputs to the element. There is also a *stability predicate* which applies to a state and an input assignment, and returns "true" if the element can stay in that state/input configuration forever. A "false" value for the stability predicate indicates that the element can stay in the configuration arbitrarily long, but must change state in finite time. Finally, there is an *output* function which maps the current internal state of the element to boolean values for its outputs.

A flow table diagram for a two-input NOR gate appears in figure 3-1. The flow table consists of two parts: on the left is the transition/stability table, with states labeling rows and input assignments labeling columns. The transition function is represented by arrows and the stable states are circled. The effect of this representation is to model a gate as an instantaneous boolean function with an *inertial* delay on its output (i.e., the case where the inputs of an inertial delay change twice before the output changes, the first change is ignored).

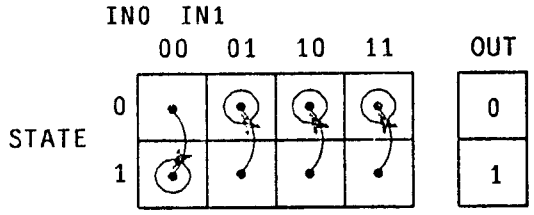

Figure 3-1: Flow Table for 2-input NOR Gate

The generality of the flow table model is useful when elements other than boolean gates are modeled. For example, many asynchronous circuits (but not our example) are designed using primitive elements other than gates (for example, flip-flops). In our example, more complex flow tables are used to model the external environment of the circuit.

Using this element model, a structural circuit description can be converted into a state graph. Each state in the global state graph is a tuple of element states, one state from each element. The global state defines a set of values for the nodes, by taking the combined effect of the output functions for each of the element states in the global state

tuple (remember that the output for each element depends only on its internal state). This gives a value for every node, since every node is connected to exactly one element output. The global state is labeled with the names of the nodes having "1" values in the state, for later use by EMC.

Given a global state and its node values, the successors of the state can be found by applying the transition functions of the individual elements. We choose to use an "interleaved" model of concurrency, in which each successor of the global state corresponds to a state change in exactly one of the elements. The global state graph for a correct circuit is often of quite reasonable size, since only the states reachable from a user-specified initial state need to be represented. This is easily accomplished by constructing the graph recursively in a depth-first manner, starting with the initial state. States are stored in a table as they are created; whenever an old state is encountered, the recursion stops.

A more subtle aspect of the element model and state-graph construction is the need for stability predicates. In order to verify "liveness" properties of a circuit, which assert that it makes progress towards some state, it is necessary to avoid checking paths in the state graph in which an element remains forever in an unstable configuration. The problem is solved using the *fairness constraint* facility of EMC. A name is invented for each state in each circuit element. Every global state is labeled with the names for the unstable element states in it, as determined by the stability predicates. In addition to the state graph, EMC is supplied with a set of fairness constraints requiring each of the unstable state names to be *false* infinitely often. In this way, CTL formulas are checked only over those paths in which no element stays in an unstable state infinitely long.

This approach has been used to discover a timing problem in a published speed-independent arbiter design [11] and to verify a corrected version of the design.

3.2. Almost Speed-Independent Circuits

The circuit model and state-graph construction described so far are sufficient to verify truly speed-independent circuits. However, many asynchronous circuits that are not strictly speed-independent work perfectly well in practice. Such circuits are often defended on the grounds that they would work properly, assuming that the delay in one element is always less than the delay in another. We can formalize a delay model which includes assumptions of this type. We call circuits designed under this delay model *almost speed-independent.*

The almost speed-independent state graph of a circuit is constructed by associating with each global state a set of relationships (called *delay constraints*) of the form $x < y$, where x and y are the names of elements that are unstable in that global state. If $x < y$ is associated with a global state, it means that "x will change state before y does." In this case, the global state will have a successor representing a change in x but no successor for a change in y. Under a speed-independent model, either x or y could change.

Two states are considered to be equivalent if their element tuples are the same *and* if their delay constraints are the same. (This is important for the table lookup in the state-graph construction.) The resulting state graph is similar to the speed-independent state graph, except paths that fail to satisfy the user-supplied delay constraints have been removed. This model and algorithm are described in greater detail in another paper [10].

3.3. Example: An Asychronous Queue Element

As an example, we verify an asynchronous queue cell patented by Cogar in 1965 [9]. The suggested application for this circuit is as a buffer for reading data from recordings. A schematic appears in figure 3-2. The original circuit diagram was depicted as a collection of diodes and transistors; we have coallesced these into NOR gates.

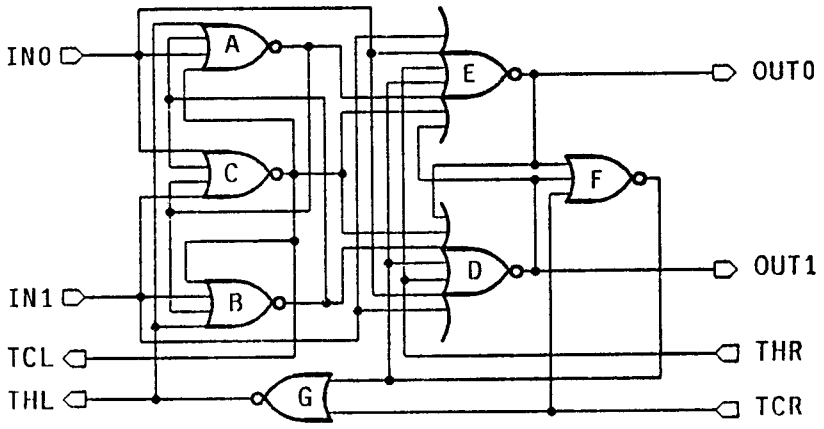

Figure 3-2: Logic Diagram for Cogar Queue Element.

The queue cells are intended to be chained together with a data source at one end and a data sink at the other. When a queue cell is empty and data is presented at its input, it reads the data and stores it internally, then makes it available at its output. In the discussion below, we consider a queue cell in the middle of a chain of queue cells in which data flows from left to right. The left neighbor of the cell can either be another queue cell or a data source; the right neighbor, either a queue cell or data sink.

The data is encoded on two wires, a "one" wire and a "zero" wire. A 1 signal is encoded as a high value on the "one" wire and a low value on the "zero" wire. Similarly a 0 is represented by a low value on "one" and a high value on "zero". "No data" is represented by a low signal on both wires, and it is never the case that both wires are high. Each queue cell has a pair of input data wires entering from the left, and output data wires exiting from the right.

In addition to the data wires, there are two control wires going from right to left. Each cell has a "transfer and hold" (TH) wire and a "transfer and clear" (TC) wire entering from the right (THR and TCR) and exiting to the left (THL and TCL). A high value on TH indicates that the cell generating it is about to become empty. A high value on TC signals that the cell is actually empty and may receive data from the cell on its left when TH goes low again.

The exact sequence of signals is shown in figure 3-3. For the interface between the cell and its left neighbor, DATA is the logical OR of IN0 and IN1, TH is THL, and TC is TCL. For the right neighbor, DATA is OUT0 ∨ OUT1, TH is THR, and TC is TCR. The heavy arrows indicate that if the first event occurs, the second must also occur – and furthermore, the events must occur in order shown. The light arrows specify that the second event may or may not occur, but if it does occur it *must* happen after the first event. The two light arrows represent the cases where the left cell provides no data, and where the right cell stops reading data.

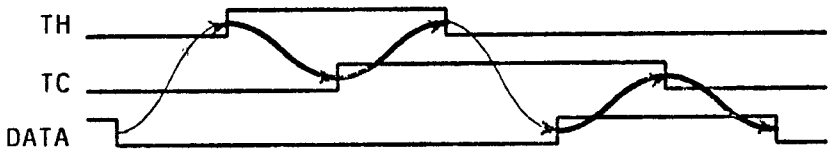

Figure 3-3: Timing Diagram for Cogar Queue Element.

This timing diagram can be expressed in CTL. Each transition can be uniquely characterized by the values of the signals immediately after it occurs. The arrows impose a total ordering on the states, which can be specified in CTL by saying, in essence, "if the signals are in the state immediately after the tail of an arrow, they must stay in that state until they take on the values of the state after the head of the arrow". The light arrows are represented by formulas using **u** and the solid arrows by U. For example, the first two arrows in the diagram are expressed by:

$$\mathbf{G}((\neg TH \wedge \neg TC \wedge \neg DATA) \rightarrow (\neg TH \wedge \neg TC \wedge \neg DATA)\ \mathbf{u}\ (TH \wedge \neg TC \wedge \neg DATA)), \quad \text{and}$$

$$\mathbf{G}((TH \wedge \neg TC \wedge \neg DATA) \rightarrow (TH \wedge \neg TC \wedge \neg DATA)\ \mathbf{U}\ (TH \wedge TC \wedge \neg DATA))$$

A queue cell can store one of three "values": 0, 1 and *empty*. If a cell is empty, its left neighbor will detect this by TCL being high. If the neighbor has a stored value, it will put the data on IN0 and IN1 of the cell. The cell then stores this value and lowers TCL. When the left neighbor sees TCL go low, it becomes empty and lowers both IN0 and IN1.

Even after it has stored data, the cell continues to supply a "no data" signal on OUT0 and OUT1 until its right neighbor has lowered THR and raised TCR. It then puts its stored value on OUT0 and OUT1. After the right neighbor has stored this, it will lower TCR, causing the cell to clear itself (and eventually remove the data signals). During the time when the cell is clearing itself, THL goes high to inhibit the transmission of data by the left neighbor since TCL may rise before the cell has stabilized in the "empty" state. THL falls after TCL has risen, to indicate that the cell is both empty and ready to receive new input.

The gate-level implementation of the queue cell appears in figure 3-2. The gates A, B, and C together store the data internal to the cell. When the circuit is stable, only one of them may be high at a time. If the high signal is A, a 1 is stored, if it is B, a 0 is stored, and if it is C, the cell is empty. The gates are cross-wired to ensure that only one is high at a time. Additionally, IN0 is an input to gates A and C and IN1 is an input to B and C, so the high input wires inhibit the gates that should not be high (thus, A and B are both inhibited when there is no data). The output of C becomes TCL, which indicates to the left neighbor that the gate is empty.

The gates D and E provide the signals OUT1 and OUT0 to the right neighbor. Once again, only one of these gates should be on at one time. The signals IN0 and IN1 are inputs to these gates, so they are both inhibited when

the cell is storing data from its left neighbor. They are also cross-wired so that only one is high at a time. The output of gate C goes to both gates, so the outputs are low when the cell is empty. Also, the output of A goes to E so that OUT0 is low when a 1 is stored; similarly, the output of B goes to D. THR is an input to both gates so that output is suppressed when the right neighbor is being cleared. Finally, the output of gate F is fed back to both of the gates. This signal is high only when both D and E are low and when TCR is low. In the case where the cell is empty, its right neighbor is full, and the cell stores an input from its left neighbor, the F signal serves to lock the cell outputs low until TCR goes high (signaling that the right neighbor has become empty).

Gate G provides the signal to clear the cell when its right neighbor has stored the cell output. Suppose that the output of gate F is high, due to the circumstances described in the last sentence of the preceding paragraph. When the right neighbor raises TCR, F will eventually go low. This allows the cell to transmit its output to the right neighbor (assuming it is not empty). When the right neighbor has stored the value, TCR will go low. This raises the output of G, which both clears the cell (by lowering A and B), and raises TCL, which turns off output of the left neighbor until the cell has stabilized.

We have not considered a number of other possible sequences of events. These might arise due to differences of speeds of the various gates when input changes are presented simultaneously to several of them. Also, changes in the input signals from the left and right neighbors can occur at many different times relative to each other and to the internal state of the cell. This task is better left to the verifier.

3.4. Verifying the Circuit

In order to verify the queue cell, we first defined flow tables for elements representing a data source and sink, which exhibit the most general behavior satisfying the above protocol. We then verify the system consisting of a single queue element with a source as its left neighbor and a sink to its right. Note that the outputs to these elements are connected to the inputs of the cell; thus, the circuit satisfies the requirement that all nodes be connected to an output. We specify a set of properties, most of which are not presented here, sufficient to ensure that the queue element would behave properly as part of a multi-cell queue.

The state graph constructed using speed-independent assumptions had 226 states. The property "if the cell is empty and IN0 is high, then IN0 stays high until 0 is stored in the cell", written

$$G((\neg A \wedge \neg B \wedge C \wedge IN0) \rightarrow (IN0\ U\ (\neg A \wedge B \wedge \neg C)))$$

fails to check (as do formulas for some of the arrows in the timing diagram). Suppose that the cell is empty, and the source provides a 0 input (rasing IN0). Gate C will go low, as it should. But this lowers TCL which signals to the source that it should lower IN0. If gate B is sufficiently slow, IN0 will disappear before B has changed state, leaving all of A, B, and C low. At this point the circuit is forced to make a non-deterministic choice between raising one of them. In reality even worse situations could occur, including a metastable state between some combination of the gates or a common-mode oscillation.

It is reasonable to defend the circuit on the grounds that the source cell to the left would not be as fast as gate B, because there would be a substantial amount of logic between the TCL signal entering the source and the gate that eventually turned off IN0. This is certainly true if the source is actually another queue cell.

What happens if the source is assumed to be slower than either A or B? Adding this assumption to the circuit description results in a state graph with 72 states that satisfies all of our specifications (including those not given here). In this case, there are fewer states in the second graph partly because of the reduced non-determinism in the circuit operation.

4. Verifying High Level Descriptions of Circuits

In practice, many circuits are designed as finite state machines before they are implemented in hardware. Since this finite state machine is close to the form of state graph used by EMC, there is no need to extract it from a lower-level description of the circuit (as in the previous section). 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.

4.1. 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 **parallel** is provided to allow several statements to execute concurrently in lockstep. The **break** statement can be used to terminate a **parallel** before all of the concurrent statements have finished executing. 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 [5]. 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:

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

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

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 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.

4.2. Example: An ACIA Controller

The best way to illustrate the use of SML is by an example. In this example, SML will be used to design some of the controlling circuitry for an Asynchronous Communications Interface Adapter (ACIA) similar to the 6850 [15].

An ACIA converts serial data to parallel data so that it can be used by a microprocessor. The serial data is a fixed number of bits that are transmitted at a known frequency, preceded by a start bit (0) and followed by one or two stop bits (1). As the data arrives, it is shifted into a data register that can be read by the microprocessor. Once a complete byte has been received or a transmission error has been detected, the ACIA sets the appropriate bits in its status register and interrupts the microprocessor if necessary. The microprocessor can read the status register to find out what happened and then clear both registers by reading the data.

In our example, the peripheral providing the data has two connections to the ACIA, *DATA* and *DCD*. *DATA* is the input data stream of 5 bit bytes, synchronized to the ACIA controller clock by another state machine. *DCD* is the data carrier detect signal that indicates that valid data is being transmitted. If this signal goes low, the transfer has been aborted by the peripheral and the microprocessor should be interrupted so that it can find the problem.

The microprocessor has three input connections to the ACIA, *COMMAND*, *C1*, and *C0*. When *COMMAND* is high, the other two inputs indicate which function the microprocessor wants to perform. The functions are:

C1	*C0*	Function
0	0	Enable/Disable interrupt when byte has been received.
0	1	Read data register.
1	0	Read status register.
1	1	Master reset.

The ACIA controller has four outputs. When the *SHIFT* signal is high, the incoming data will be shifted into

the data register. Once a complete byte has been received, the *RDRF* (read data register full) status bit should be set high. *RDRF* should be cleared if the ACIA is reset, the carrier is lost, or if the microprocessor reads the data after reading the status and finding *RDRF* high. The *IRQ* output is connected to the microprocessor to signal an interrupt request. The request is cancelled if the ACIA is reset or if the microprocessor reads the status and data registers. The *DCDO* (data carrier detect outstanding) status bit is set high whenever there has been a loss of carrier that hasn't been handled by the microprocessor. It is set high as soon as the *DCD* input goes low. It remains high as long as *DCD* remains low. If the microprocessor either reads the status and data registers or resets the ACIA while *DCD* is low, *DCDO* will go low as soon as the carrier is restored. Otherwise, *DCDO* will remain high until the status and data registers are read or the ACIA is reset.

4.3. An Implementation of the ACIA Controller in SML

```
1       program acia;
2
3       input C0, C1, COMMAND, DCD, DATA;
4       output SHIFT, RDRF, IRQ, DCDO;
5       internal NEWIRQ, DCDST, DCDRD, RIE;
6
7       #define RESET           (COMMAND & C1 & C0)
8       #define READSTAT        (COMMAND & C1 & !C0)
9       #define READDATA        (COMMAND & !C1 & C0)
10
11      procedure wait(exp)
12          while !(exp) do loop skip endloop
13      endproc
14
15          parallel
16              loop
17                  parallel
18                      loop
19                          if !DCD | RESET then break endif
20                      endloop
21                  ||
22                      loop
23                          wait(!DATA);
24                          raise(SHIFT);
25                          delay 4;
26                          lower (SHIFT);
27                          parallel
28                              if RIE then
29                                  parallel raise(IRQ) || raise(NEWIRQ) endparallel
30                              endif
31                          ||
32                              raise (RDRF)
33                          endparallel
34                      endloop
35                  endparallel;
36                  parallel lower(SHIFT) || lower(RDRF) endparallel
37              endloop
38          ||
39              loop
40                  wait(!DCD);
41                  parallel
42                      raise(DCDO)
43                  ||
44                      if !RESET then
45                          parallel raise(IRQ) || raise(NEWIRQ) || raise(DCDST) || raise(DCDRD) endparallel
46                      endif
47                  ||
48                      wait(DCD)
49                  endparallel;
50                  DCDO := DCDRD;
51              endloop
52          ||
53              loop
54                  wait(COMMAND);
55                  switch
56                      case READSTAT:
57                          parallel lower(NEWIRQ) || lower(DCDST) endparallel;
58                          break;
59                      case READDATA:
60                          parallel
61                              lower(RDRF) || IRQ := NEWIRQ || DCDRD := DCDST
62                          endparallel;
63                          break;
64                      case RESET:
65                          parallel lower(IRQ) || lower(NEWIRQ) || lower(DCDST) || lower(DCDRD) endparallel;
66                          break;
67                      default:        -- change read interrupt enable
68                          invert(RIE);
69                          break;
70                  endswitch
71              endloop
72          endparallel
73      endprog
```

Figure 4-1: A First Attempt at Writing the ACIA Controller in SML

Figure 4-1 shows a program that attempts to implement the ACIA controller. 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.

Lines 3-5: In addition to declaring the input and output signals, four internal booleans are also declared. *NEWIRQ* is true when an interrupt has been generated but the microprocessor has not discovered why by reading the status register. *DCDST* is true when the carrier has been lost but the microprocessor hasn't read the status register to discover the problem. *DCDRD* is true when the carrier has been lost and the microprocessor has not handled this by reading the status register and then the data register. *RIE* (receive interrupt enable) is true if the microprocessor should be interrupted when a complete byte has been received.

Lines 7-9: The SML compiler includes a preprocessor similar to the C preprocessor. These lines define three macros that are used to decode the command that the microprocessor is issuing.

Lines 11-13: **Wait** is defined to be a macro that does nothing until its argument becomes true.

Lines 22-34: This loop handles the incoming data. The procedure is to wait for the start bit (line 23) and then start shifting (line 24). After 4 more bits have been received, stop shifting (lines 25 and 26). The data register is now full, and an interrupt should be sent if interrupts are enabled (lines 27-33). The loop is restarted, and the ACIA waits for the next start bit.

Lines 18-20: At any time, it is possible for the carrier to go away or the microprocessor to execute a reset. Should this happen, the current transfer should be aborted. This loop waits for either event to occur, and then breaks out of the **parallel**, thus terminating the execution of the loop at lines 22-34.

Line 36: If the carrier is lost or the ACIA should be reset, the break on line 19 causes execution to resume here. This statement stops shifting and empties the data register. Then the normal incoming data loop is restarted.

Lines 39-51: This loop handles the loss of carrier. To begin with, the controller waits for the carrier to go away (line 40). Once the carrier has been lost, the *DCD* status bit is set (line 42). If the ACIA is not being reset, the ACIA interrupts the microprocessor and indicates that a new loss of carrier has been detected (lines 44-46). Then the ACIA waits for the carrier to be restored (line 48). Once the carrier has been restored, the *DCD* status bit is cleared (line 50) if the microprocessor has done something about the loss of carrier (and *DCDRD* has been reset as a result).

Lines 53-71: This loop handles requests from the microprocessor. Nothing is done until a valid command is received (line 54). If the command is a read status, the microprocessor will know about any interrupt or loss of carrier, so *NEWIRQ* and *DCDST* are cleared (lines 56-58). If the command is a read data, the data register has been emptied, so RDRF should be cleared (lines 59-63). In addition, if a prior read status has found out about an interrupt (*NEWIRQ* is low), the interrupt should be cancelled. If a prior read status has found out about a loss of carrier (*DCDST* is low), then we should indicate that the loss has been handled by lowering *DCDRD*. If the command is a reset, the ACIA cancels any interrupt and ignores a simultaneous loss of carrier (lines 64-66). If the command is enable/disable interrupts, the current value of *RIE* is changed (lines 67-69). After processing the command, the loop is restarted and the ACIA waits for the next command.

This program was compiled into a minimal 450 state machine in approximately 25 seconds of CPU time on a VAX.

4.4. Verifying the ACIA Controller Using the Model Checker

The correct behavior of a controller such as the ACIA is typically described by a set of timing diagrams, such as the one in figure 4-2. In this section, we describe how this timing diagram can be expressed in CTL.

In order to simplify the CTL description of this timing diagram, we introduce a set of macro definitions. Since *DCD*, *DCDO* and *IRQ* are usually high and *COMMAND* is usually low, we define **normal** to be $DCD \wedge DCDO \wedge IRQ \wedge \neg COMMAND$. In addition, **readstat** is defined to be $COMMAND \wedge C1 \wedge \neg C0$ and **readdata** is defined to be $COMMAND \wedge \neg C1 \wedge C0$. Using these simple definitions, we introduce a macro **nextcommand**(*type,next*) that is true if along all paths where *DCD* remains high and the next microprocessor request is *type*, *DCDO* and *IRQ* remain high and *next* becomes true after the command is received. The definition of **nextcommand** is:

$$\textbf{nextcommand}(type,next) := [\textbf{normal}\ \mathbf{u}\ \neg\textbf{normal} \wedge (DCD \wedge type \rightarrow DCDO \wedge IRQ \wedge \mathbf{X}\ next)]$$

In English, this formula states that along all paths, *DCD*, *DCDO* and *IRQ* are high and *COMMAND* is low until something changes. At this point, if *DCD* is still high and the microprocessor has issued a *type* command, then *DCDO* and *IRQ* should remain high and *next* should be true in all next states.

Now that we have defined **nextcommand**, it is a simple matter to describe this timing diagram in CTL by the formula:

$$\mathbf{G}\,(\neg DCD \wedge \neg COMMAND \wedge \neg DCDO \wedge \neg RIE \rightarrow$$

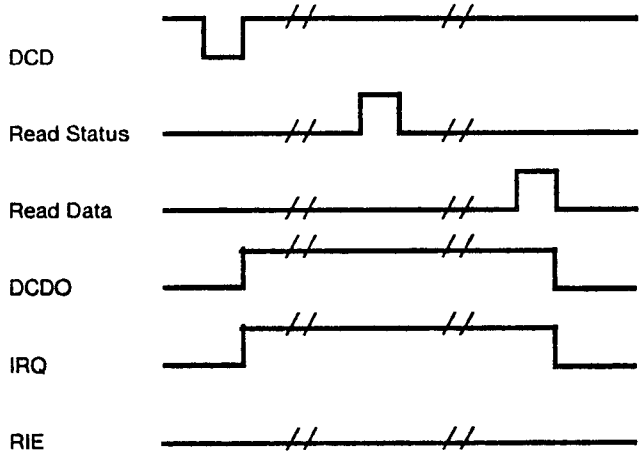

Figure 4-2: A Sample Timing Diagram for the ACIA

```
% emc -c acia1.asm
                CTL MODEL CHECKER (version B1.0)

Reading acia...
Fairness constraint: .

time:  (1066 244)

|= G (~DCD -> X DCDO).
The equation is TRUE.


time:  (338 89)

|= G (~DCD & ~COMMAND & ~DCDO & ~RIE ->
      X nextcommand(readstat, nextcommand(readdata, ~DCDO & ~IRQ))).
The equation is FALSE along the path:
State 0:
State 1: C1 COMMAND DCD NEWIRQ DCDST DCDRD IRQ DCDO
State 377: C0 COMMAND DCD DCDRD SHIFT IRQ DCDO
State 260: COMMAND DCD SHIFT DCDO

time:  (2392 536)

|= .
End of session.
%
```

Figure 4-3: Verifying the First ACIA Controller Program

```
% emc -c acia2.asm
                CTL MODEL CHECKER (version B1.0)

Reading acia...
Fairness constraint: .

time:  (813 252)

|= G (~DCD -> X DCDO).
The equation is TRUE.


time:  (276 66)

|= G (~DCD & ~COMMAND & ~DCDO & ~RIE ->
      X nextcommand(readstat, nextcommand(readdata, ~DCDO & ~IRQ))).
The equation is TRUE.


time:  (1670 581)

|= .
End of session.
%
```

Figure 4-4: Verifying the Corrected ACIA Controller Program

X nextcommand(*readstat*, nextcommand(*readdata*, $\neg DCDO \wedge \neg IRQ$)))

At the begining of figure 4-2, *DCD*, *COMMAND*, *DCDO*, and *RIE* are all low. Whenever this situation occurs, *DCDO* and *IRQ* will go high and remain high until a read status command is received (this is the meaning of **nextcommand**(readstat...)). After the read status, *DCDO* and *IRQ* will remain high until a read data command is received (according to the nested **nextcommand**). After the read data, *DCDO* and *IRQ* will both go low.

Figure 4-3 shows a transcript of the model checker running on the program in figure 4-1. The numbers in parentheses are the total user cpu time and "system time", in $1/60^{th}$s of a second.

As the transcript shows, the program does raise *DCDO* whenever *DCD* goes high, but it doesn't lower *DCDO* if *DCD* goes low before a read data occurs. Therefore, we change lines 59-63 of the original program to be:

```
59                      case READDATA:
60                          parallel
61                              lower(RDRF) || IRQ := NEWIRQ || DCDRD := DCDST || DCDO := DCDST | !DCD
62                          endparallel;
63                          break;
```

With this change, *DCDO* will be lowered if *DCD* is high and *DCDST* is low (indicating that a read status has occurred). The new program compiles into 352 states in approximately 20 seconds of CPU time. The correctness of this program is shown by the transcript in figure 4-4.

5. Directions For Future Research

Some circuits seem easier to specify by using a timing diagram like figure 3-3 than by a CTL formula. Of course, CTL 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 CTL formulas or check them directly using an algorithm similar to the one used by EMC. 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.

In the case of asynchronous circuits more research is needed on realistic timing models. Clearly the arbitrary delay model used in [11] is too conservative. The use of delay constraints as described in section 3 is a step in the right direction. But it still does not handle timing assumptions like the 3/2 rule, which requires that the delay through any three gates be greater than the delay through any two. A more general scheme for delay constraints that will handle the 3/2 rule and other similar timing assumptions is under development.

More research is also needed on how to extend the approach described in this paper to large, hierarchically constructed circuits. A first step in this direction is discussed in [8]. If one uses a subset of CTL, lower level circuits can be simplified by "hiding" some of their internal nodes (more precisely, making it illegal to use them in CTL formulas) and merging groups of states that become indistinguishable into single states.

Finally, we believe in practice it may be possible to verify circuits that consist of arrays of identical cells by using the model checker to verify a generic instance of the cell design and then applying some form of induction. Because CTL is propositional, however, the induction rule will probably have to be formulated outside of the logic.

References

[1] M. Browne, E. Clarke, D. Dill, B. Mishra, "Automatic Verification of Sequential Circuits", *CHDL85*, Tokyo, August 1985.

[2] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics", Tech. report, Ecole Nationale Superieune des Mines de Paris, 1984.

[3] M. Ben-Ari, Z. Manna, A. Pneuli, "The Logic of Nextime", *8th ACM Symposium on Principles of Programming*, Jan. 1981.

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

[5] M.C.Browne, E.M.Clarke, "SML-A Finite State Language".

[6] E.M. Clarke, E.A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", in *Proc. of the Workshop on Logic of Programs, Yorktown Heights, NY*, 1981.

[7] E.M. Clarke, E.A. Emerson, 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, 1983.

[8] E.M. Clarke and B. Mishra, "Automatic Verification of Asynchronous Circuits", in *Proc. of Logics of Programs, Pittsburgh. Pa.*, E. Clarke and D. Kozen, eds., 1983.

[9] George Cogar, "Asynchronous Self Controlled Shift Register", U.S. Patent Office 3,166,715, January, 1965.

[10] David L. Dill, "Verification of Asynchronous Circuits with Timing Dependencies".

[11] David L. Dill and Edmund M. Clarke, "Automatic Verification of Asynchronous Circuits using Temporal

Logic", *1985 Chapel Hill Conference on VLSI*, Computer Science Press, May 1985.

[12] E.A. Emerson, E.M. Clarke, "Characterizing Properties of Parallel Programs as Fixpoints", in *Proc. of the Seventh International Colloquium on Automata, Languages and Programming*, 1981.

[13] Y. Malachi and S. S. Owicki, "Temporal Specifications of Self-Timed Systems", *VLSI Systems and Computations*, 1981.

[14] A. Pneuli, "The Temporal Semantics of Concurrent Programs", *18th Symposium on Foundations of Computer Science*, 1977.

[15] Veronis, Andrew, *Microprocessors: Design and Applications*, , 1978.