Automatic verification of asynchronous circuits using temporal logic

D.L. Dill, M.S., and E.M. Clarke, Ph.D.

Indexing terms: *Circuit theory and design*

Abstract: A method is presented for automatically verifying asynchronous sequential circuits using temporal logic specifications. The method takes a circuit described in terms of Boolean gates and Muller elements, and derives a state graph that summarises all possible circuit executions resulting from any set of finite delays on the outputs of the components. The correct behaviour of the circuit is expressed in CTL, a temporal logic. This specification is checked against the state graph using a 'model checker' program. Using this method, a timing error in a published arbiter design is discovered. A corrected arbiter is given and verified.

1 Introduction

Any sequential digital circuit built from nontrivial subcircuits must ensure that timing constraints among the subcircuits are satisfied. The most widely used technique for doing this is synchronous circuit design, where a global signal (e.g. a clock) is shared among all the subcircuits. The use of such a signal simplifies circuit design, as the speeds of component circuits do not need to be considered, provided the clock is slow enough.

An alternative is to use asynchronous design. Instead of a global signal, some other means is used to determine when a subcircuit computation is complete. Asynchronous design is more difficult than synchronous design and is much less widely used.

This is unfortunate. Distributing a global clock signal that must appear almost simultaneously at all parts of a circuit becomes increasingly more difficult as more components are packed onto VLSI chips. Asynchronous designs can reduce this problem by being more modular; for example, a subcircuit can provide an explicit completion signal only to the other circuits that use its outputs.

Additionally, asynchronous circuits are sometimes faster than synchronous circuits. Suppose that one circuit computes a result that is used by another. In an asynchronous design the second circuit can proceed immediately when the result from the first circuit is available. In a synchronous design the second circuit must wait for a clock signal, which is timed to assume the worst-case delay for the first circuit.

Why is asynchronous design so difficult? Because the circuit may perform different sequences of actions depending on the relative delays of its components, delays that may not be known or may even vary with time. When even a few components are operating concurrently, the circuit designer must consider a large number of cases to be sure that the circuit works properly. This is a tedious, counterintuitive and error-prone process.

Methods to debug and check combinational and sequential circuits are not as helpful for asynchronous circuits. Building a working prototype does not adequately assure correctness, as component delays in other instances of the same circuit may differ from those of the prototype. The same problems plague all existing simulators: They can only simulate the circuit under a small set of assumed delays in the components and input signals.

This paper describes a general method for verifying asynchronous circuits. We propose that the correct behavior of a circuit be specified using *temporal logic*. The specification can be checked automatically against a gate-level description of the circuit. If no violation of the specification is detected, the circuit designer can be confident that the circuit will perform according to the specification for all relative gate delays. If the circuit fails to meet its specification, the verifier provides a counter example to help the designer isolate the problem.

We also describe an application of this technique to a published design for an asynchronous arbiter. The verifier discovered a set of relative delays which could cause the circuit to fail. A corrected version of the same circuit has been successfully verified using the same specification.

In Section 2 of this paper we describe the language for specifying the correct behaviour of circuits. It is a *propositional temporal logic*, a logical system for reasoning about the ordering of conditions in time. This Section also describes a program, the *model checker*, that checks temporal logic formulas against a state graph.

Section 3 describes the functional behaviour of the example arbiter. This Section also presents various parts of the temporal logic specification of the arbiter and explains them informally. For example, we specify that the arbiter interfaces obey the four-cycle signalling convection, that the arbiter provides proper mutual exclusion between two users and that the arbiter is responsive to requests.

Section 4 explains the gate-level circuit descriptions. The circuit is written as a collection of primitive components connected with wires. The predefined primitive components are Boolean gates and two additional components: Muller C and ME elements (the second provides mutual exclusion). A flow-table semantics for these primitives is described that allows for arbitrary finite delays on the gate outputs. The state graph representing a circuit, which can be used as input to the model checker of Section 2, is defined using these flow tables.

Section 5 describes the gate-level implementation of the original arbiter. The verification of the arbiter and the timing problem we discovered are discussed. We give a corrected circuit that can be verified.

We conclude with a discussion of the role we envision for this technique and some directions for future research.

2 Logic and model checker

To verify a circuit, we need a language in which to specify the correct behaviour. We have chosen to use a *propositional temporal logic* called CTL. Propositional logic is well established as a good formalism for specifying digital systems. CTL is traditional propositional logic with additional connectives for talking about the future states of a

Paper 4759E (E2), first received 25th April and in revised form 5th December 1985

The authors are with the Department of Computer Science, Carnegie-Mallon University, Pittsburgh PA 15213, USA

circuit. The additional connectives provide a means for specifying sequential behaviour.

There is a program for checking CTL specifications against a finite state graph, called EMC (extended model checker). If the CTL formula is false in the state graph, EMC will show a counterexample, if possible, in the form of a sequence of states that violates it [3].

EMC is a general purpose CTL verifier. There are several preprocessors that translate various special-purpose representations into state graphs which are used as input to EMC. There are translators for a CSP-like language for describing finite-state programs, a language for behavioural descriptions of synchronous sequential circuits, and a gate-level language for structural descriptions of asynchronous circuits. The results reported here were obtained using the last of these.

In this paper we use a subset of CTL and a somewhat different notation from that actually used in the model checker. We now describe the syntax and semantics of our dialect. We assume the existence of an underlying set of atomic propositions AP. The formulas of CTL are

(i) any atomic proposition $p \in AP$ is a CTL formula

(ii) if f and g are CTL formulas $\neg f$, $f \wedge g$, $f\mathbf{U}g$, $\mathbf{G}f$, and $\mathbf{F}f$ 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 $\neg(f \wedge \neg g)$. Intuitively, $f\mathbf{U}g$ (read 'f until g') means that f must always be true until g becomes true. It is permissible for g never to be true if f remains true forever: this is a 'weak' rather than a 'strong' until. $\mathbf{G}f$ means that f is true in every state (globally true), and $\mathbf{F}f$ means that f is eventually true somewhere along every sequence of states (true at some future time).

The semantics of CTL are defined more precisely in terms of *truth* in a *structure*. A CTL structure is a state graph consisting of:

(i) a finite set of states S

(ii) a function $L \in [S \rightarrow \mathscr{P}(AP)]$, which labels the states with atomic propositions

(iii) a transition relation $R \in S \times S$

(iv) a start state, s_0.

We require that every state have at least one successor.

A *path from* s_i, where $s_i \in S$, is an infinite sequence of states that starts with s_i and in which R holds between every state and its successor. We define formally 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* $p \in L(s_i)$ (for $p \in AP$)

$s_i \models f \wedge g$ *iff* $s_i \models f$ and $s_i \models g$

$s_i \models \neg f$ *iff* $s_i \nvDash f$

$s_i \models f\mathbf{U}g$ *iff* for every path from s_i either (i) there exists a state s_k such that $s_k \models g$ and for all states s_j preceding s_k, $s_j \models f$, or (ii) for all states s_j on the paths, $s_j \models f$

$s_i \models \mathbf{G}f$ *iff* for every state s_j on every path from s_i, $s_j \models f$.

$s_i \models \mathbf{F}f$ *iff* there exists a state s_j on every path from s_i such that $s_j \models f$.

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

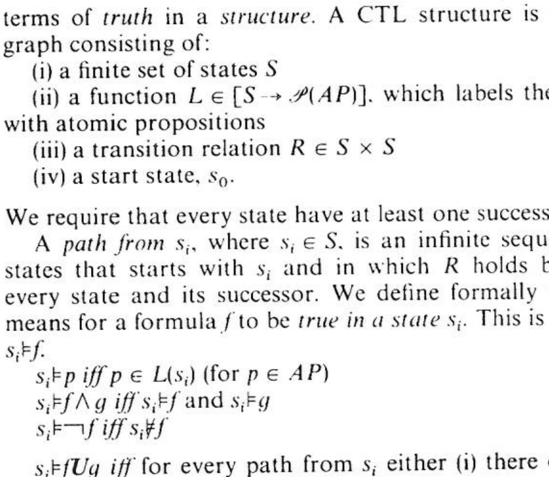

Fig. 1 *CTL formulas and state graphs in which they are true*

EMC is a program that checks whether a formula is true in a state graph. 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. In practice, EMC can check formulas on graphs with several hundred states in less than 10 seconds.

Frequently, we want to consider only *fair* execution sequences. For example, we may wish to consider only those executions in which some process that is continuously enabled eventually fires. The semantics of CTL have been modified slightly to allow this. The user can provide a set of *fairness constraints*, each of which is an arbitrary CTL formula. A path is defined to be pair with respect to a set of fairness constraints if each constraint holds infinitely often along the states of the path. The path quantifiers are restricted to fair paths. The addition of fairness constraints does not significantly reduce the speed of EMC [3]. In Section 4, we use fairness constraints to limit the fair paths to those in which all gates exhibit *finite* switching delays.

3 Specification of the arbiter

This Section explains the function of the arbiter and gives a temporal logic specification for it. An arbiter grants a resource to no more than one of several 'users' in case they request the resource simultaneously. A typical use of such a device is to negotiate access to a memory shared among several processors. The arbiter design given here was originally proposed by Seitz [8]. Seitz's design was later specified with temporal logic and verified by hand by Bochmann [1], who found some potential timing errors resulting from misprints in the original paper. The design we present includes Bochman's modifications to correct these problems.

The arbiter is intended to co-ordinate two users (see Fig. 2). When it grants the resource to one of the two users, it first activates a 'transfer module' associated with that user. The transfer module prepares parameters for a single 'resource module' (representing the shared resource) which is activated after the transfer module has finished.

Many of the verification conditions which follow appear in some form in the Bochman paper mentioned before. There is no direct correspondence because different logics and circuit semantics are used.

We only require that the arbiter meet its specification when it is properly used. In particular, all circuits connected to the arbiter must conform to a specific signaling convention. Each module explicitly signals requests and completion by use of *four-cycle signalling* (also called *Muller signalling*). There are five four-cycle interfaces in the block diagram of Fig. 2.

In this protocol, two wires, r (request) and a (acknowledge), are used to connect a *client* module to a *server* module. (These are generic signal names; we leave it to the reader to substitute the actual signal names for each of the interfaces.) Initially the signal on both wires is 0. The client makes a request by raising r. Eventually the server acknowledges the request by raising a. The client then lowers r and the server lowers a. At this point the interface is back in the original state awaiting a request from the user (see Fig. 3).

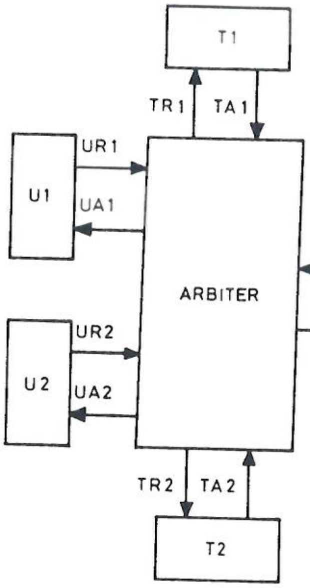

Fig. 2 *Arbiter block diagram*

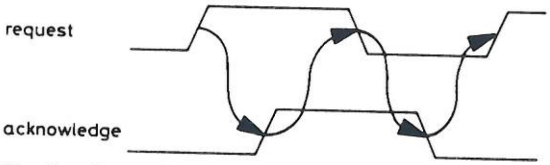

Fig. 3 *Four-cycle timing diagram*

We do not require the user modules, U1 and U2, to make a request, but we do require that they respond to an acknowledge: When the server raises a in response to r going high, the user must inevitably lower r. The server modules, T1, T2 and S, must inevitably raise and lower a in response to the raising and lowering of r.

When the arbiter acts as a server (as it does for U1 and U2), it must satisfy

$$\mathbf{G}[a \rightarrow (a\mathbf{U}\neg r)] \tag{1}$$

and

$$\mathbf{G}[\neg a \rightarrow (\neg a\mathbf{U}r)] \tag{2}$$

When it is acting as a client (as with T1, T2, and S), it must satisfy

$$\mathbf{G}[r \rightarrow (r\mathbf{U}a)] \tag{3}$$

and

$$\mathbf{G}[\neg r \rightarrow (\neg r\mathbf{U}\neg a)] \tag{4}$$

We also wish to ensure that once the arbiter has acknowledged a request, or has had a request acknowledged, the rest of the four-cycle convention will inevitably be carried out:

$$\mathbf{G}[(r \wedge a) \rightarrow \mathbf{F}(\neg r \wedge \neg a)] \tag{5}$$

In the remainder of this Section, we sometimes omit trailing 1s and 2s on signal names. Any formula with such names represents two formulas: one in which 1 has been appended to all the short names and one in which 2 has been appended.

We require that every request to T1 or T2 be properly motivated by a request from U1 or U2; it is unacceptable for there to be an activation of T1 or T2 without a request from the corresponding user module, or for there to be multiple activations of T1 and T2 to service a single request. Furthermore, we require that the users wait if they have not been granted the resource; the arbiter must not acknowledge a user unless the arbiter has committed to activating the corresponding T module.

$$\mathbf{G}[(\neg ur \wedge \neg aa) \rightarrow \neg tr] \tag{6}$$

$$\mathbf{G}[\neg tr \rightarrow (\neg tr\mathbf{U}ur)] \tag{7}$$

$$\mathbf{G}\{(ur \wedge tr) \rightarrow [tr\mathbf{U}(\neg tr\mathbf{U}\neg ur)]\} \tag{8}$$

and

$$\mathbf{G}[(\neg ur \wedge \neg ua) \rightarrow (\neg ua\mathbf{U}tr)]. \tag{9}$$

The first formula says that tr is 0 if there is no user request in progress. The second requires that tr stay 0 until there is a user request. (At first, it may appear that this formula is implied by the first formula. It is not: The second formula covers the case where ua is high because a previous request has not been completely processed.) The third formula states that if tr has gone to 1 in response to ur going to 1, then tr will not go to 0 and to 1 again until after ur has gone to 0 (at most one request to the appropriate T module for each user request). The final condition requires the arbiter not to acknowledge the user until tr has been raised (i.e. the resource has been granted to the user). (These four properties are not given in the Bochman paper.)

We assume that the T modules compute inputs for S, and that the arbiter must wait until T has finished computing these signals before sending a request to S. We also assume that the T modules signal the completion of the parameter computation by raising ta and that T will keep the parameters stable until tr is lowered. Hence, the arbiter must raise tr, wait until T raises ta, raise sr and wait until S raises sa before lowering tr. These conditions are expressed by

$$\mathbf{G}\{\neg sr \rightarrow [\neg sr\mathbf{U}(ta1 \vee ta2)]\} \tag{10}$$

and

$$\mathbf{G}[tr \rightarrow (tr\mathbf{U}sa)] \tag{11}$$

These conditions also imply that requests to S cannot occur without a request to T1 or T2, and that a new request to T1 or T2 cannot occur until S has dealt with the previous request on that module.

The essential property of the arbiter is that at most one of U1 and U2 may be granted the resource at the same time. We write this as

$$\mathbf{G}\{tr1 \rightarrow [\neg tr2\mathbf{U}(\neg tr1 \wedge \neg sa)]\} \tag{12}$$

and

$$\mathbf{G}\{tr2 \rightarrow [\neg tr1\,\mathbf{U}(\neg tr2 \wedge \neg sa)]\} \tag{13}$$

With the exception of the last four-cycle condition, none of the properties so far requires the arbiter to make progress at any point in processing a request; they merely require that if the arbiter *does* attempt to respond to a request, it does so correctly. We would like to be able to show that every user request will eventually be acknowledged

$$\mathbf{G}[(ur \wedge \neg ua) \rightarrow \mathbf{F}ua) \tag{14}$$

Unfortunately, it seems that proving this property for arbiters similar to this example requires stricter timing assumptions than we use. It always appears to be possible for one user to be denied service forever because the arbiter choses to process repeated requests from the other user. It may be able to verify this property using another circuit model (e.g. a model that gives explicit delays for various components), but for this model we must be satisfied with the weaker requirement that *one* of the requesting users will inevitably be served

$$G\{[(ur1 \wedge \neg ua1) \vee (ur2 \wedge \neg ua2)] \rightarrow F(ua1 \vee ua2)\} \qquad (15)$$

Coupled with the four-cycle convention and the requirement that a user not be acknowledged until it is served, this guarantees that some user will inevitably be served. Note that if U2 stops making requests after being served, requests from U1 will inevitably be served. Thus, preemption by U2 is the only way that U1 can be permanently denied service.

4 Circuit semantics

This Section discusses the formal model of gate-level circuits used in the verification technique. Our model is a modified version of the flow tables traditionally used in the analysis of asynchronous sequential circuits.

At any time a circuit has a set of input values, an internal state and a set of output values. If the input values change, the circuit may change to another state after an aribtrary delay. The output signals reflect the internal state *immediately*. Thus, Boolean gates act like Boolean functions with arbitrary nonzero delays on the outputs.

The input to the circuit-to-state-graph translator is a syntactic description of the circuit. The description gives the names of the wires and gates and indicates how the wires are connected to the inputs and outputs of the gates. Each of the gates has a prototypical flow table which is instantiated by substituting the actual wire names for the appropriate inputs and outputs. This process is straightforward and is not discussed further. The flow tables are then 'glued together' into a global state graph.

Given a finite set W of names of wires, a flow table consists of

(*a*) a set of input wires $I \subseteq W$

(*b*) a set of output wires $O \subseteq W$

(*c*) a set of states C

(*d*) a transition function $T \in [\![[I \rightarrow \{0, 1\}] \times C \rightarrow \mathscr{P}(C)]\!]$

(*e*) a stability predicate $P \in [\![[I \rightarrow \{0, 1\}] \times C \rightarrow \{t, f\}]\!]$

(*f*) an output function $N \in [\![C \rightarrow [O \rightarrow \{0, 1\}]]\!]$.

For every $x \in [I \rightarrow \{0, 1\}]$ and every $y \in C$ we require that $y \in T(x, y)$. This provides the arbitrary delay property as the circuit always has the option of staying in its current state.

Flow tables can be summarised in diagrams such as those in Fig. 4, which gives the complete set of flow tables used in the arbiter example. The left-hand box in each diagram represents the transition function: Input assignments are labelled on the top, and states (numbered arbitrarily) are labelled on the left side. Each square in this box contains the set of possible successor states, given the current state and input values corresponding to the square (every state is implicitly a successor to itself). There is a circle in a square if the stability predicate is true for that input assignment and state. The output box has states along the left-hand side and outputs along the top.

These differ from traditional flow tables in the meaning of the stability predicate. A particular combination of an input assignment and a state is stable if and only if the circuit is allowed to stay in that configuration forever. If a configuration is not stable, the circuit may stay in that configuration arbitrarily long, but the delay must be *finite*.

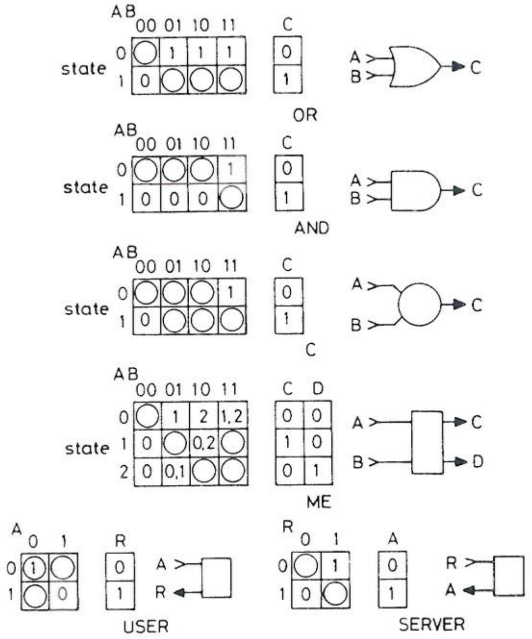

Fig. 4 *Flow tables*

Hence, a configuration which has no exiting transitions is necessarily stable. A configuration with exiting transitions may or may not be stable. If it is not stable, the circuit must eventually take one of these transitions unless the input changes; if the configuration is stable, the circuit is allowed either to stay in it forever or to exit.

The stability predicate is needed to enforce reasonable circuit behaviour. For example, an AND gate with inputs 00 and output 1 must change values eventually; otherwise many obvious circuit properties are impossible to verify.

Another novel aspect of this flow-table model is that it allows simultaneous signal changes in inputs and outputs. Most discussions of asynchronous circuits assume that only one signal can change at any time. Although it is unreasonable for the correct behaviour of an asynchronous circuit to depend on two signals changing simultaneously, a general verification technique should be able to discover failures that can only occur in the case of simultaneous signal changes.

In Fig. 4, the tables for AND and OR gates should be self-explanatory. The *C element* is often used in asynchronous circuits to wait for the completion of all of several concurrent operations before proceeding to the next operation. The circuit exhibits hysteresis: When the inputs disagree, the circuit holds its current output and does not change until all of the inputs have values opposite to the output.

The heart of the arbiter is the *ME element* (mutual exclusion element, also called an *interlock*). This circuit never allows both of its outputs to be high. When the inputs are both low, so are the outputs. When one of the inputs is high, the corresponding output will eventually go high. When an input and the corresponding output are both high, the output stays high until the input goes low. Most interestingly, when both inputs are high and the outputs are both low, the ME element has the option of raising either, but not both, of the outputs. This is an additional source of non-determinism in the arbiter besides the varying speeds of the gates. Also, note that the outputs of the ME element can go to either 00 or 01 when the inputs go from 10 to 01.

Another interesting circuit is the USER. This circuit simulates the most general behaviour of a user module. We can verify a USER circuit only when the interfaces behave reasonably (in this case, they must obey the four-cycle protocol). Unlike the others, this circuit has a transition leaving a *stable* state. With this table, the USER can either generate a request or not (note that the user *must* eventually lower the request if it has been acknowledged). Aside from the stability predicate, this table is identical to that for an inverter.

To verify a circuit, we must be able to combine the behaviours of the primitive components of the circuit into a state graph. We are given a description of the circuit as a set of primitive components (in this case the components of Fig. 4). The connections between the components are encoded in the input and output sets for each component: Inputs and outputs that are wired together have the same name. In a well-formed circuit one gate output at most can be connected to a set of inputs, although any number of inputs can be connected.

To build the state graph, we further require that *every* input be connected to an output. Any circuit can be converted to this form by adding circuits to simulate input sources. We provide circuits to simulate U1, U2, T1, T2 and S in our example.

We define a large state graph from which we remove irrelevant states to give the actual state graph. We index the k component circuits by $1 \leq i \leq k$. In general, we refer to the parts of component i by subscripting their names (e.g. the transition function for component i is T_i).

We define a *state vector* to be a vector of length k which has a member of C_i as its ith element, for all i. We take the set of states S of the state graph to be the set of all state vectors. A vector of component states is designated to represent the start state, s_0 (the user determines an initial state for the circuit).

Given a function $f \in [S \rightarrow T]$, we write the restriction of f to $S' \subset S$ as $f|_{S'}$. Let $\boldsymbol{s}$ be any state vector, and let s_i be its ith element (a member of C_i). $\boldsymbol{s}$ determines $v \in [W \rightarrow \{0, 1\}]$ such that $v|_{O_i} = N_i(s_i)$ for all components i. This represents the combined outputs of the component circuits. v is unique because every wire is connected to an output. The state is labelled with the names of the wires to which v assigns 1.

We can use the wire values v and the transition functions T_i for the individual components to find the successors in the global state graph. Formally, $v|_{I_i}$ gives an input assignment for each component i. The set of successors to $\boldsymbol{s}$ is the set of all state vectors $\boldsymbol{s}'$ such that, for all i, $s'_i \in T_i(v|_{I_i}, s_i)$, where s'_i is the ith element of $\boldsymbol{s}'$. This defines R, the transition relation for the state graph.

This state graph has many irrelevant states. The final equivalent, and much smaller, state graph is obtained by restricting it to the states reachable from s_0.

Remember that each flow-table state is a successor of itself. In any state there is a set of active components that have the option of changing state. Because of the nature of the computation of R there is a successor state vector for every combination of components that changes state and every component that stays in the same state. In this way the state graph represents all possible relative delays between changes of component states.

The algorithm to build the state graph starts with an initial state vector, then builds the graph in a recursive, depth-first manner. As each state is created it is stored with its state vector in a hash table. For each new state the algorithm finds the combined output function (v, above), the successor states for each component and the set of state vectors for the successors.

There is one additional problem in the state graph. Every state has itself as a successor (from the case where all component circuits stay in the same state). If we want to be able to verify properties such as the last formula of Section 3, we need to force the model checker not to consider paths in which a component stays in an unstable configuration forever. For this we find fairness constraints to be necessary.

For each state in each component machine i we invent a unique label l. Let $\boldsymbol{s}$ be the state vector for any state, let s_i be its ith element, and let v be the combined output function, as defined above. We label $\boldsymbol{s}$ with l if and only if $P_i(v|_{I_i}, s_i) = f$.

We then supply a fairness constraint to the model checker that requires l to be *false* infinitely often. In this way, the *fair* paths are those in which a gate must leave an unstable state infinitely often. The model checker will only consider these paths when checking a formula.

We have recently become aware of work by Muller in which a similar notion of fairness, using generalised regular expressions, is used to capture the semantics of arbitrary yet finite delays in infinite circuit executions (although the term 'fairness' is not used) [7]. Our solution is somewhat more general than Muller's, as it allows infinite delays in states that have outgoing transitions (such as the USER element) and applies to circuit elements other than Boolean gates (such as the C and ME elements). Of course, the application to circuit verification is new.

5 Implementation and verification of the arbiter

In this Section, we present and explain the implementation of the arbiter from Boolean gates and Muller elements, describe the verification of the circuit and the problem it detected and present a corrected circuit.

. The circuit shown in Fig. 5 was originally presented by Seitz, with some errors in transcription [8]. The errors were detected by a verification attempt by Bochmann [1], and the corrections were described there. This circuit incorporates the corrections described by Bochmann.

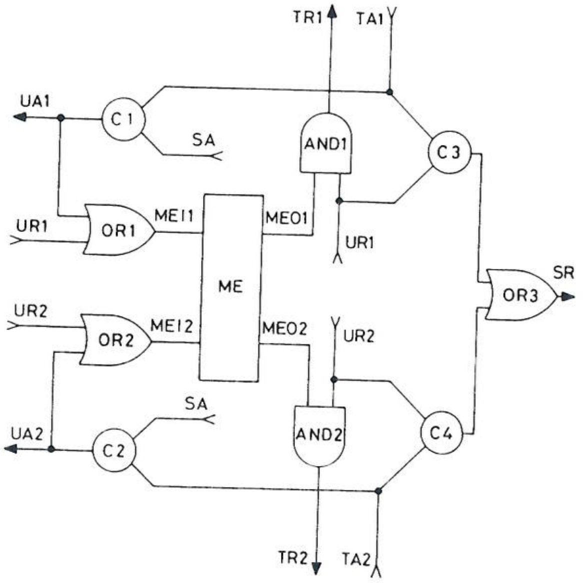

Fig. 5 *Original arbiter implementation*

Initially, the verification of this circuit ran into difficulties because the global state graph was very large. When the program ran out of memory, there were more than 10000 states in the partially constructed state graph.

Before abandoning this approach, we tried to verify the model with only U1 making requests, by substituting a dummy circuit for U2. Any failure detected under these conditions would be a failure under the more general situation where both users make requests. The resulting state graph was manageable (160 states). To our surprise, the model checker detected a failure in the circuit even under these conditions.

The error is a violation of the four-cycle interface with T1. Suppose that the arbiter starts in the state where all nodes are 0 and responds to a request from U1. After the arbiter has almost completed processing this request, it is possible for it to be in the state where all nodes are low except *meo*1, which remains high (due to a long delay in the ME element). If at this point U1 raises *ur*1 again, AND1 will raise *tr*1. Now, if *mei*1 has not yet risen in response to *ur*1 (because of a long delay in OR1), it is possible for *meo*1 to fall and for AND1 to respond by lowering *tr*1. But *ta*1 never goes high! EMC detects a sequence of states exhibiting this behaviour, and gives it as a counterexample to formula 3 $(\boldsymbol{G}[r \rightarrow (rUa)])$.

Charles Seitz has indicated to us that the arbiter was designed with the assumption that the delay between *ua* going high and *ur* going low would be long relative to the internal signals of the arbiter. This assumption was not stated in the original paper due to an oversight. Although we have not attempted to verify the arbiter with this constraint, we believe that it would meet our specifications.

How did Bochmann successfully verify this circuit? He used a more forgiving circuit model: Boolean gates were assumed to respond immediately to their inputs, and the output of the ME gate was assumed to fall immediately when its input falls. Under these assumptions our failing scenario does not occur because *meo*1 could not be high when *ur*1 is raised again. (In fact, the CTL versions of verification conditions given by Bochmann are true in the state graph he gives for the circuit.) Our model is more strict: for this design it is not *too* strict because the design was intended to be speed independent.

Seitz has supplied us with several alternative designs that are correct according to our specifications. The circuit of Fig. 6 is the simplest of these. This simplicity is gained at the cost of some concurrency (and probably also speed): The 'return to zero' transitions of the four-cycle interfaces with the T1 or T2 and S modules are intended to be concurrent in the first circuit, but in this circuit *ta*1 must go to zero before *sr* does. The gates AND1 and AND2 prevent the arbiter from triggering both T modules in some circumstances. Consider the case where U1 has requested the resource. Eventually, the arbiter will reach a state where *ua*1 has been raised (*ta*1 and *sa* will be high). If *ur*1 goes low and *ur*2 goes high before *ua*1 goes low, AND2 prevents *tr*2 from being signalled. This prevents a variety of unfortunate events, including, possibly, a violation of the four-cycle interface with the S module.

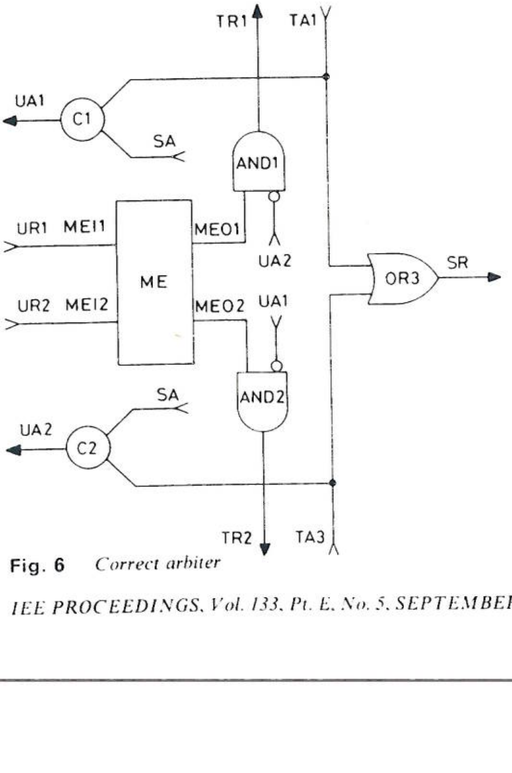

Fig. 6 *Correct arbiter*

The modified circuit has been fully verified using the techniques here. Interestingly, even with both users intact, the state graph for the second circuit consists of only 62 states. EMC required about 3 s to check formula 15. It is interesting to note that correct arbiters with more components than the original have state graphs with fewer than 200 states [4]; we have often noticed that correct circuits have state graphs of quite reasonable size, whereas incorrect circuits have enormous state graphs.

It took less than a minute to construct the global state graph from the circuit description; we believe that this could be improved considerably by eliminating low-level implementation inefficiency (e.g. our Lisp system calls a procedure for every array access).

6 Conclusions and future research

We have presented a practical method for assuring the correctness of small speed-independent circuits. We believe that being able to verify small circuits is valuable in its own right. As we hope to have demonstrated with the aribiter example, small circuits can both be useful and difficult. Furthermore, a likely approach for larger circuits is *multilevel verification*, which would have verification of small gate-level subcircuits as one component.

Perhaps a more orthodox approach would have been to use temporal logic to describe the circuit components (in addition to using it as a specification language). We prefer our approach for two reasons: first, although temporal logic is appropriate for specifications of large circuits, we feel that flow tables provide a more natural way to describe the operation of small circuit elements than temporal logic. Secondly, the relevant decision procedures for temporal logic are substantially more complicated that the algorithms for constructing and checking models, and, consequently, are likely to be slower in practice.

However, it would be desirable to use the same formalism for describing circuit components and for specifications. This would simplify hierarchical verification of large, modular circuits: Once the 'important' properties of a circuit had been verified, the circuit could be treated as a primitive in larger circuits, using its specification as its description. We are currently investigating the use of automata for circuit descriptions and for specifications. We hope that this work will eventually yield a more uniform, compositional formalism, perhaps in the spirit of CCS [5].

Another area for further work is that of alternative timing models. The arbitrary gate delay model is very conservative. The use of more liberal timing models could result in more economical circuits, both in time and area. One such model is the 'almost equal delay' model, in which a ratio is specified between the delay of the slowest gate and the delay of the fastest gate [2].

Finally, there is the problem of assuring the correctness and completeness of a specification. We see no full solution to this problem, although we hope that some guidelines for important properties to check will emerge when there has been more experience with this type of verification. As we hope our example has demonstrated, automatic verification can be a powerful debugging technique even when there is no guarantee of the correctness of the logical specification.

7 Acknowledgments

We would like to thank Chuck Seitz for providing several examples of correct arbiters and for comments on our verification results. Randy Bryant and Mary Sheeran provided valuable comments on earlier drafts of this paper. We are also grateful to Chris Hanna for technical editing of an earlier version of this paper.

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

8 References

1 BOCHMANN, G.V.: 'Hardware specification with temporal logic: an example', *IEEE Trans.*, 1982, **C-31**, (3), pp. 223–231

2 BRZOZOWSKI, J.A., and YOELI, M.: 'Digital networks' (Prentice-Hall, 1976)

3 CLARKE, E. M., EMERSON, E. A., and SISTLA, A. P.: '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, pp. 117–126

4 DILL, D.L., and CLARKE, E.M.: 'Automatic verification of asynchronous circuits using temporal logic', 1985 Chapel Hill Conference on VLSI, pp. 127–143

5 MILNER, R.A.: 'Calculus of communicating systems', Lecture Notes in Computer Science 92 (Springer-Verlag, 1980)

6 MISHRA, B., and CLARKE, E. M.: 'Automatic and hierarchical verification of asynchronous circuits using temporal logic', CMU-CS-83-155, Department of Computer Science, Carnegie-Mellon University, September 1983

7 MULLER, D.E.: 'The general synthesis problem for asynchronous digital networks', Conference Record of the Eighth Annual Symposium on Switching an Automata Theory, 1967, pp. 71–82

8 SEITZ, C.L.: 'Ideas about arbiters', *LAMBDA*, First Quarter, 1980, pp. 10–14

9 WOLPER, P.: 'Temporal logic can be more expressive', Proceedings of the 22nd Symposium on Foundations of Computer Science, October 1981, pp. 340–348