Hardware Specification with Temporal Logic: An Example

GREGOR V. BOCHMANN

***Abstract*—The use of temporal logic for the specification of hardware modules is explored. Temporal logic is an extension of conventional logic. While traditional logic is useful for specifying combinational circuits, it is shown how the extensions of temporal logic apply to the specification of memory, as well as the safeness and liveness properties of active circuits representing processes. These ideas are demonstrated by the example of a self-timed arbiter. An implementation of the arbiter is also given, and its formal verification by a kind of reachability analysis is discussed. This verification approach is also useful for finding design errors, as demonstrated by an example.**

***Index Terms*—Arbiter, design verification, hardware specification, hardware verification, logic design, modal logic, self-timed systems, temporal logic, VLSI design.**

I. Introduction

As systems become more and more complex, good methods for specifying the systems and their submodules become more and more important. In this paper we explore the possibility of using temporal logic, an extension to traditional logic and predicate calculus, for the specification of hardware modules. While traditional logic is useful for specifying system states that are possible at some given time, temporal logic provides additional facilities for specifying possible state sequences, such that the evolution of the system may be characterized.

Although it seems that considerations of real time may be added to the temporal logic formalism, we do not deal with such considerations in this paper. We focus our attention to systems in which the submodules interact asynchronously through some appropriate signaling scheme. In such systems we are not (so much) interested in the question at exactly what time some thing will happen, but rather that they *will eventually* happen. The formalism of temporal logic contains constructions for speaking exactly about these considerations [7].

We note that temporal logic has first been proposed for the specification of software systems [1]. In fact, the design of most software modules does not require real-time considerations, but only considerations for *eventual* (and not "too slow") response. This is also true for asynchronous interaction between hardware components. Such interaction has been used in many systems. It has been argued recently [2] that asynchronous interaction between components within VLSI systems allows more flexibility in the design by using "self-timed" submodules. The specification method described in this paper is directly applicable to such systems at the circuit level of detail. Specification at a somehow more abstract level is discussed in a companion paper [3].

We give in this paper first an overview of temporal logic, and introduce the **while** operator which we found particularly useful for describing memory aspects of hardware modules. We give a formal meaning to this operator by defining proof rules in the context of a transition model of computation. The main part of the paper then deals with the example of a self-timed arbiter, as described in [4]. This part describes the properties of the arbiter, its implementation in terms of more primitive logical circuits, and an analysis of this implementation covering up an error and some "difficulties." The last section contains some remarks on the use of temporal logic and conclusions.

Manuscript received June 30, 1980; revised May 27, 1981. This work was supported in part by the Computer Systems Laboratory, Stanford University, Stanford, CA, DARPA Contract MDA903-79-C-0680.

The author is with the Département d'Informatique et de Recherche Opérationnelle, Université de Montreal, Montreal, P.Q., Canada.

II. Temporal Logic

A. Informal Introduction

We give in the following a very informal introduction to temporal logic. A more precise discussion of this topic may be found in [1]. We assume that the reader is familiar with traditional logic, as far as Boolean algebra and simple predicate calculus is concerned. While the traditional logic uses such operators as $\wedge$, $\vee$, $\Rightarrow$, $\neg$, etc., temporal logic introduces additional operators for dealing with temporal sequences. While an expression of the traditional predicate calculus is assumed to specify properties of the system state at some given time, which in the following is called the "present" time, an expression of the temporal logic is assumed to specify properties of all possible execution sequences that may evolve from the present system state. We explain in the following the typical temporal operators.

The "Henceforth" Operator $\Box$: The expression $\Box A$ means that the assertion A is true at the present time and at all future times. For example, the assertion

$$B \Rightarrow \Box A$$

means that whenever B is true at some particular time (which is said to be the "present" time), then A is true also and will remain forever. We note that the assertion $\Box A$ is equivalent to A, and means that A is true at each time that may be considered present, i.e., at all times. This also means that A is an invariant.

The "Eventually" Operator ∇: The expression ∇A means

0018-9340/82/0300-0223$00.75 © 1982 IEEE

that the assertion A will be true at some future time, possibly the present time, but not necessarily remain true. For example, the assertion

$$B \Rightarrow \nabla A$$

means that if B is true in the present time, i.e., at some given time, then A will eventually become true. This operator is usually used to specify the eventual response of a module to some given request.

Temporal operators may be combined. For example, the assertion

$$B \Rightarrow \nabla \Box A$$

means that if B is true at present then there will eventually be a time from which on A will always remain true. (It may have been true intermittently before.) The assertion

$$B \Rightarrow \Box \nabla A$$

means that if B is true at present, then at every future instant of time A will eventually be true after that instant; therefore A will be true infinitely often, possibly without any change.

*The **next** Operator:* The expression $B \Rightarrow$ **next** A means that if B is true at present, then A will be true at the "next" instant of time to be considered. This introduces the concept of discrete time and the concept of a transition that occurs between subsequent time instants. The same model of transitions and discrete time is often used for the description of parallel processes [5], [6]. These concepts are also the basis for the definition of the **while** operator below.

It is important to note that the use of the **next** operator represents an idealization. Using this operator for hardware specification, we assume that the transition times of the circuits considered are short compared to the time intervals of the **next** operator. This means that the next transition of the input signals only occurs when the output signals have assumed a stable value. We note that more than one input may change between two consecutive time instants considered.

B. The *while* Operator

We introduce the **while** operator with the following (informal) meaning.

$$A \textbf{ while } B$$

means that if B is true at present, then A is true at present and remains true as long as B remains true. However, it implies nothing for any time after B has been false at some given instant. More formally, we may define the **while** operator recursively using the **next** operator as follows. *Definition of the **while** operator:*

$$(A \textbf{ while } B) \text{ is equivalent to } (B \Rightarrow (A \wedge \textbf{next } (A \textbf{ while } B))).$$

In order to be able to analyze a system that is described with the help of a **while** operator, we need proof rules that define the properties of the **while** operator to be used in the logical reasoning about the described system. The following proof rules relate properties of the present state with properties at the next time instant. They are related to the transition model introduced by the **next** operator. More powerful proof rules could be derived. However, the following ones seem to be sufficient in many cases.

In order to distinguish between the properties of the present state and those at the next instant, we use in the following simple names to refer to variables and logical expressions that characterize the *present system state*, and primed names to refer to expressions that are true in the *next state*. We have the following proof rules.

(Proof 1) $$\frac{A \textbf{ while } B}{B \Rightarrow A}$$

(Proof 2) $$\frac{A \textbf{ while } B}{B \wedge B' \Rightarrow A'}$$

(Proof 3) $$\frac{A \Rightarrow A \textbf{ while } B}{A \wedge B \wedge B' \Rightarrow A'}.$$

These rules are easily derived from the definition of the **while** operator given above, noting that B' is equivalent to **next** B, and similarly for A. Clearly, (A **while** B) implies $B \Rightarrow A$. It also implies $B \Rightarrow$ **next** $(B \Rightarrow A)$, which is equivalent to $(B \wedge B' \Rightarrow A')$.

Many of the assertions involving the **while** operator in the example below are of the particular form $A \Rightarrow A$ **while** B, for which the last rule applies. It is interesting to note that this rule does not imply any property for the present state, as the first proof rule for the expression A **while** B does. It only implies a property that relates the present state to the next one. It is therefore particularly suitable for describing memory properties.

C. Comments on the Use of Temporal Logic for Hardware Specification

The use of the different operators of temporal logic is demonstrated by the arbiter example discussed in the Section III below. In this section we only wish to give the reader a general feeling for which aspects of a hardware specification the different operators of temporal logic are most suitable.

Combinational Circuits: Combinational circuits have essentially no time dependence. The output values are a function of the input values at any given instant, independently of previous values. Properties of such functions may be specified by traditional logic using input–output assertions. Temporal operators are not needed. We note that this description abstracts from such details as signal propagation and switching delays within the circuit, which may be important when the real timing of the transitions within a sequential circuit are considered.

Memory Elements: Memory elements have two aspects, namely the setting of a new value and the keeping of this value. These two aspects may naturally be described by a partially defined input–output function and a **while** clause, respectively. For example, a simple memory element that may be reset and set to an input value, may be defined as follows. The memory element interacts with its environment through the circuits RESET, SET, IN, and OUT. The following assertions hold for the values of these circuits:

RESET $\Rightarrow$ — OUT (output value set to zero)

SET $\Rightarrow$ OUT = IN (the output value is set to the input value)

OUT $\Rightarrow$ OUT **while** $\neg$ RESET $\wedge$ $\neg$ SET (keeping the value)

— OUT $\Rightarrow$ $\neg$ OUT **while** $\neg$ RESET $\wedge$ $\neg$ SET (keeping the value).

The first two formulas specify the input–output function of the memory element partially (not for the case that $\neg$ RESET $\wedge$ $\neg$ SET). The last two **while** clauses may be rewritten in the abbreviated form

$$(\text{OUT} = x) \Rightarrow (\text{OUT} = x) \textbf{ while } \neg \text{RESET} \wedge \neg \text{SET}$$

where universal quantification is assumed over the variable x taking on the Boolean values *true* or *false*.

Processes: While combinational circuits and memory elements may be considered passive components, we consider *processes* to be those system components that initiate system state changes. In many cases state changes are only initiated by a given process in response to state changes initiated by some other part of the system at some earlier instant in time (as, for example, by the arbiter considered below).

Properties of processes are often classified into two kinds, namely safeness assertions (often called "partial correctness") and liveness assertions [7]. The safeness assertions specify properties that hold at any given time (invariant assertions) and properties that determine in which order different events of state change may occur. As above, these properties may be specified by traditional invariant assertions in predicate calculus and while expressions, respectively. Other examples are the mutual exclusion property of the arbiter discussed below, and the safeness property of the four-cycle signaling scheme used by that arbiter.

The liveness assertions specify properties that imply some kind of progress and state changes that lead to some desired behavior or to some termination state. These properties may be specified using the "eventually" operator ∇, sometimes in conjunction with the "henceforth" operator $\square$.

III. The Example of an Arbiter

We consider the example of an arbiter module (adapted from [4]) which determines the order in which two user modules obtain access to a shared resource module, as shown in Fig. 1. The arbiter interacts with the other modules over circuit pairs that follow the four-cycle (FC) signaling scheme explained below. The signaling scheme determines in which form requests and acknowledgments are exchanged between two interacting modules. As shown in Fig. 1, the arbiter module interacts through five circuit pairs with the other modules. Over the pairs ($UR1$, $UA1$) and ($UR2$, $UA2$), the user modules $U1$ and $U2$, respectively, request an access to the resource. When the arbiter grants access to one of the users, it requests the transfer of some parameter values from the selected user to the resource through the corresponding pair ($TR1$, $TA1$) or ($TR2$, $TA2$). When the parameters have been transferred the arbiter sends a request to the resource through the pair (SR, SA). After the resource has acknowledged the request, the cycle may start again, possibly granting access to the other user.

A. Specification

1) Specification of the Four-Cycle Signaling Scheme: Before we give a precise specification of the arbiter module, we define now the properties of the four-cycle signaling scheme used for the interaction between the modules. The scheme is

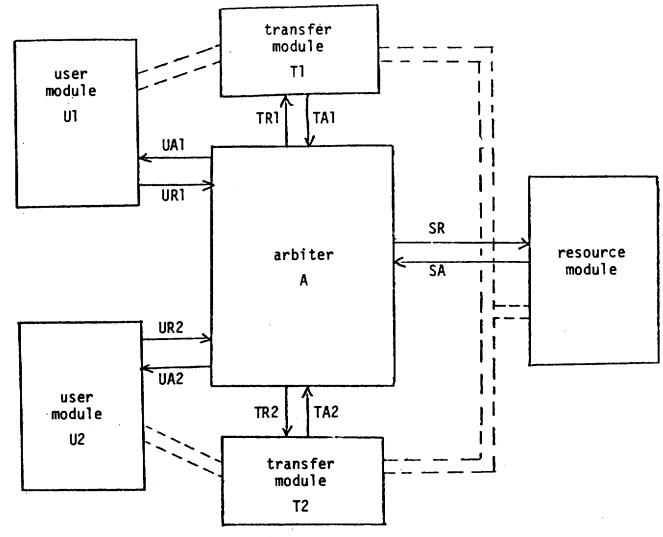


Fig. 1. The interactions of the arbiter module with the other modules in the system.

used over a pair of circuits, which we name R and A in the following. The R circuit carries the *request* from one module (which we call the *requesting* module) to the other (which we call the *responding* module). The A circuit carries the acknowledgment back from the responding module to the requesting one. The order in which these circuits may change their value is governed by the four-cycle scheme, which is informally shown in Fig. 2. Using the notation in Section II, we may define this signaling scheme as follows:

($FC1$) $(R = x) \Rightarrow (R = x)$ **while** $(A = \neg x)$

($FC2$) $A \Rightarrow \nabla \neg R$.

These two assertions should be satisfied by the *requesting* module. ($FC1$) states that the request circuit should not change its value until the acknowledge circuit has assumed the same value. For instance, when a request has been made (R set to one) the request circuit must stay up until an acknowledgment has been received (A set to one). And a new request may only be given after or when the acknowledgment signal is reset.

Assertion ($FC2$) states that the requesting module should eventually terminate the request after the acknowledgment has been received.

($FC3$) $(A = x) \Rightarrow (A = x)$ **while** $(R = x)$

($FC4$) $R \Rightarrow \nabla A$

($FC5$) $\neg R \Rightarrow \nabla \neg A$.

These assertions should be satisfied by the *responding* module. ($FC3$) corresponds to ($FC1$) and states that the acknowledge circuit may only change its value when the request circuit has the opposite value. ($FC4$) and ($FC5$) state that such a change of A must eventually occur some time after the request circuit R has changed.

2) Specification of the Arbiter: We give now a precise specification of the behavior of the arbiter module, using the notation explained in Section II. The specification consists of three parts:

1) an enumeration of the interface circuits used for interaction with other modules,

2) assumptions about the behavior of the other modules with which the arbiter interacts, and

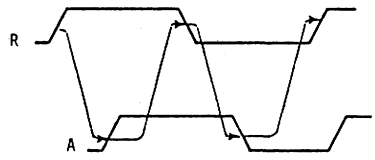


Fig. 2. Time diagram of the four-cycle signaling scheme.

3) assertions that must be satisfied by the arbiter provided the assumptions are met.

Point 1) is explained above (see Fig. 1); the other points are given below.

Assumptions:

$$(UiFC1) \text{ and } (UiFC2) \qquad \text{for } i = 1, 2.$$

These assumptions are obtained from $(FC1)$ and $(FC2)$, respectively, by replacing R by URi and A by UAi. For example, $(U2FC2)$ reads "$UA2 \Rightarrow \nabla - UR2$."

$$(TiFC3), (TiFC4), \text{ and } (TiFC5) \qquad \text{for } i = 1, 2.$$

They are obtained from $(FC3)$, $(FC4)$, and $(FC5)$ by replacing R by TRi and A by TAi.

$$(SFC3), (SFC4), \text{ and } (SFC5).$$

They are obtained from $(FC3)$, $(FC4)$, and $(FC5)$ by replacing R by SR and A by SA.

For example, the assumptions $(UiFC1)$ and $(UiFC2)$ mean that the user modules Ui [which are *requesting* modules for the circuit pairs (URi, UAi), $(i = 1, 2)$] satisfy the four-cycle signaling conventions. (We note that somehow stronger assumptions are actually needed, as discussed in Section III-D3.) The other assumptions, similarly, apply to the parameter transfer and resource modules (which are *responding*).

Assertions:

$(UiFC3)$, $(UiFC4)$, $(UiFC5)$, $(TiFC1)$, $(TiFC2)$, $(SFC1)$, $(SFC2)$.

These assertions are derived from the four-cycle assertions (FCk), $k = 1, \cdots, 5$ similarly as the assumptions above. They state that the requirements of the four-cycle signaling scheme must be satisfied by the arbiter, as a *responding* and a *requesting* module, respectively.

$$(A1) \qquad \neg(TR1 \wedge TR2).$$

This assertion states the mutual exclusion property that only one parameter transfer may be requested at any given time.

$$(A2) \quad SR \wedge \neg SA \Rightarrow (TR1 \wedge TA1) \vee (TR2 \wedge TA2).$$

$$(A3) \qquad TRi \Rightarrow TRi \textbf{ while } \neg SA \ (i = 1, 2).$$

These additional safeness assertions may be required depending on the interaction between the parameter transfer and the resource modules. It may be necessary that the parameter transfer request is acknowledged and still active when the resource receives the request $(A2)$, and remain active until the resource gives its acknowledgment $(A3)$.

$$(A4) \qquad URi \Rightarrow \nabla (TRi \wedge SR) \qquad \text{for } i = 1, 2.$$

This assertion states the liveness property of the arbiter, namely that after a request is given by the user module Ui, eventually this request will be served by transferring the user's parameters and making a request to the shared resource. We note that the arbiter implementation satisfies even the stronger assertion

$$(A5) \quad URi \Rightarrow \nabla (URi \wedge TRi \wedge SR) \qquad \text{for } i = 1, 2$$

which implies, together with the other assertions, that the user request will be acknowledged only after it has been passed to the resource.

B. Implementation of the Arbiter

Fig. 3 (adapted from [4]) shows a possible implementation of an arbiter in terms of more primitive submodules. The submodules used here are elementary combinational AND and OR circuits, as well as more complex Muller *rendezvous*(C) and *mutual exclusion* (ME) circuits, which are discussed in more detail in [2] and [4], respectively.

The rendezvous circuit is defined as follows. When both inputs of a C-element assume the same value, then its output will be the same value. This is the only occasion when the output may change. This property may be defined by the following assertion:

$$(C) \quad (\text{IN1} = \text{IN2} = x) \Rightarrow (\text{OUT} = x) \textbf{ while } ((\text{IN1} = x) \vee (\text{IN2} = x))$$

An equivalent pair of assertions is

$$(C1) \quad (\text{IN1} = \text{IN2} = x) \Rightarrow (\text{OUT} = x)$$

$$(C2) \quad (\text{OUT} = x) \Rightarrow (\text{OUT} = x) \textbf{ while } ((\text{IN1} = x) \vee (\text{IN2} = x)).$$

The properties of a ME-element may be defined as follows (for a possible implementation of such a ME-element, see, for example, [4]):

$$(ME1) \qquad MEOi \Rightarrow MEOi \textbf{ while } MEIi.$$

This means that when an output is up it remains up as long as the corresponding input is up.

$$(ME2) \qquad \neg(MEOi \wedge \neg MEIi).$$

This means that an output is only up while the corresponding input is also up.

$$(ME3) \qquad \neg(MEO1 \wedge MEO2).$$

This states the mutual exclusion between the two outputs.

$$(ME4) \quad \Box MEIi \wedge \Box \nabla \neg MEOj \Rightarrow \nabla MEOi \ (i, j = 1, 2; i \neq j).$$

This assertion states that if a given input $MEIi$ remains up forever and the opposite output is down forever, or an unlimited number of times, then the output corresponding to the given input will eventually come up. This is a relatively weak fairness assumption. A stronger assumption is

$$(ME4^*) \quad \neg(MEIi \wedge \neg MEIj \wedge \neg MEOi)(ij = 1, 2; i \neq j)$$

which states that whenever there is a request on input $MEIi$ and no request on the other input then the corresponding output $MEOi$ must be up. The weaker assumption will be used in the proof of the arbiter liveness in Section III-D4.

C. Verification of the Arbiter Implementation

A verification of the implementation consists of showing that the assertions of the arbiter specification given in Section III-A2 can be derived from the assertions for the submodules

out of which the arbiter is built (see above), taking into account the way these submodules are connected with one another and to the interface circuits of the arbiter.

We give in this section some informal arguments that the arbiter implementation is correct by considering all possible states into which the operation of the implementation may lead (from some assumed initial state), showing that all states and transitions satisfy the arbiter assertions. This is an informal *reachability analysis*. Starting in the initial state of the implementation, this analysis considers, from each intermediate reachable state, all possible transitions that may lead to other states. The possible transitions are induced by value changes of input circuits to the arbiter, as well as by value changes generated within the arbiter by some of its submodules. (In the case of this arbiter implementation, the only internally generated change may come from the ME-element for which an output may go up some undetermined time after the corresponding input went up [4]; this is expressed by the *eventually* operator ∇ in assertion ($ME4$). The output of the combinatorial circuits and the C-elements only change when their input change.)

As initial state we assume the value zero for all circuits. The only possible transitions from this state are due to requests from the users $U1$ and/or $U2$. These requests will set the corresponding input to the ME-element to one, but only one output of the ME-element will go up because of its mutual exclusion property. Assuming that the output $MEO1$ goes up, the table below shows the subsequent state changes. We note that the first line of the table (called "state" 1) represents actually four different states, depending whether the request circuits $UR1$ and $UR2$ are up, or not. Each of the following lines represents two individual states, depending whether the circuit $UR2$ is up, or not. The possible transitions are shown in Fig. 4 (solid arrows), where the states $s'(s = 2, \cdots 6)$ are similar to the states s, corresponding to the case that the ME-element chooses to set the other output $MEO2$ to one.

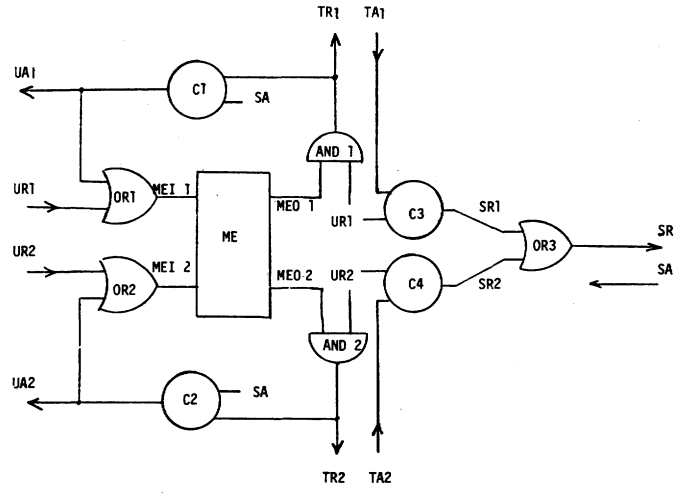


Fig. 3. An implementation of the arbiter module.

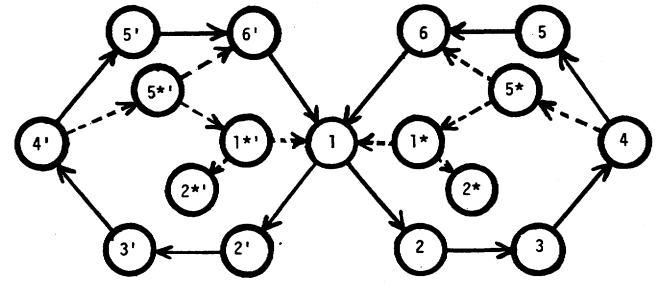


Fig. 4. Some possible transitions for the arbiter implementation.

simple sequence of transitions that may occur, which are initiated by the following events:

- $TA1$ goes up in response to the transfer request,
- SA goes up in response to the service request,
- the user request goes down,
- $TA1$ goes down after $TR1$ went down,
- SA goes down after SR went down.

State	$UR1$	$MEI1$	$MEOi$	$TR1$	$TA1$	$SR1$	SR	SA	$UA1$	Comments
1	?	?	0	0	0	0	0	0	0	Initial state
2	1	1	1	1	0	0	0	0	0	ME output has gone up
3	1	1	1	1	1	1	1	0	0	Transfer module has given acknowledgment
4	1	1	1	1	1	1	1	1	1	Resource module has given acknowledgment
5	0	1	1	0	1	1	1	1	1	User has reset the request circuit
6	0	1	1	0	0	0	0	1	1	Transfer module has reset its acknowledge circuit; transition to state 1 when the resource module resets its acknowledge circuit
5*	0	1	1	0	1	0	0	1	1	Resource module has reset its acknowledge circuit
1*	?	?	0	0	1	0	0	0	0	Transfer module still has not reset its acknowledge circuit
2*	1	1	1	1	1	0	0	0	0	ME output has gone up
6*	0	1	1	0	0	0	0	1	1	Transfer module has reset its acknowledge circuit
6**	0	1	1	0	1	0	0	0	1	Resource module has reset its acknowledge circuit

After the ME output $MEO1$ has gone up, there is only a

This last transition leads back to the state 1, with another

request possibly pending from user $U2$. Since this is the only possible sequence of transitions, it is easy to check that the safeness properties of the arbiter are satisfied. For the liveness properties of the arbiter, we refer the reader to the discussion in Section III-D4.

The same kind of analysis is also useful for finding design errors in implementations. As an example we consider again the arbiter specification in Section III-A2, but a different implementation, the one given by [4, Fig. 9]. This figure differs from Fig. 3 (except for notation) only by the fact that the Ck ($k = 3, 4$) circuits are replaced by AND gates. Making the same kind of analysis as above, we find that up to state 4, there is no change in the behavior of the system. However, from state 4, two transitions are possible. First the transition to state 5 as before, and second the transition to state 5* (see table), which occurs when the resource resets its acknowledge circuit before this is done by the transfer module. (In the implementation of Fig. 3, the former has to wait for the latter.) Assuming the transfer module is "slow," transitions to states 1* and 2* may occur before the transfer module resets its acknowledge circuit (as shown in Fig. 4, dashed arrows). But the transition to state 2* is in conflict with the four-cycle signaling scheme for the transfer circuits $TR1$ and $TA1$, since a new request is given before the acknowledge circuit is reset. We conclude that the implementation does not satisfy the signaling scheme for the interaction with the transfer module (i.e., assertion ($T1FC1$), and is therefore incorrect.

C. L. Seitz pointed out to me that this error is due to a misprint, and that the correct Fig. 9 of [4] should show the horizontal wires from $C1$ and $C2$ (using our labeling) going to $TA1$ and $TA2$, not to $TR1$ and $TR2$. Making the same kind of analysis as above, we find the same behavior as the faulty implementation up to state 5*. However, the following state is either 6* or 6**, depending on the relative speed of the transfer and resource modules. One may say that the acknowledgment circuits of these two modules are reset "simultaneously." The initial state 1 is reached as soon as both circuit have been reset.

D. Formal Verification

We now outline a formal proof that the arbiter implementation as given in Section III-B satisfies the arbiter specification given in Section III-A2. This proof is structured into the following sections:

1) rewriting the properties of the submodules of the implementation, taking into account their interconnection,

2) writing down invariant assertions that hold for all reachable system states,

3) proving these assertions,

4) deriving the safeness properties of the arbiter, and finally,

5) deriving the liveness properties of the arbiter.

1) Properties of the Submodules and their Interconnection: The following assertions are directly implied by the properties of the submodules of the implementation, as defined in section III-B, taking into account their interconnection as defined in Fig. 3.

The C-elements (for $i = 1, 2$ and $k = 3, 4$, respectively):

($Ci1$) $(TRi = SA = x) \Rightarrow (UAi = x)$

($Ci2$) $(UAi = x) \Rightarrow (UAi = x)$ **while** $((TRI = x) \vee (SA = x))$

($Ck1$) $(TAi = URi = x) \Rightarrow (SRi = x)$

($Ck2$) $(SRi = x) \Rightarrow (SRi = x)$ **while** $((TAi = x) \vee (URi = x))$

The OR-gates (for $i = 1, 2$):

(ORi) $MEIi = URi \vee UAi$

(OR3) $SR = SR1 \vee SR2$

The AND-gates ($i = 1, 2$):

(ANDi) $TRi = MEOi \wedge URi$

The ME-element:

The assertions ($ME1$) through ($ME4$) given in Section III-B.

2) Invariant Assertions: As shown in the next section, the following assertions (for $i = 1$ and 2) hold for all reachable states of the arbiter implementation.

($I1$) $SRi = TAi$

($I2$) $SR = SR1 \vee SR2$

($I3$) $SA = UA1 \vee UA2$

($I4$) $\neg MEOi \Rightarrow \neg TRi \wedge \neg SRi \wedge \neg UAi$

($I5$) $MEOi \Rightarrow TRi = URi$

($I6$) $MEOi \Rightarrow$
$(TRi \wedge \neg TAi \wedge \neg UAi)$
$\vee (TRi \wedge TAi \wedge \neg UAi)$
$\vee (TRi \wedge TAi \wedge UAi)$
$\vee (\neg TRi \wedge TAi \wedge UAi)$
$\vee (\neg TRi \wedge \neg TAi \wedge UAi)$

($I7$) $\neg (MEO1 \wedge MEO2)$.

We define classes of system states (similarly, as in Section III-C) by the following assertions:

(State 1): $\neg MEO1 \wedge \neg MEO2$

(State k) ($k = 2, 3, 4, 5, 6$): $MEO1 \wedge \neg MEO2 \wedge$ the (k-1)th subclause of ($I6$) with $i = 1$

(States k') ($k = 2, 3, 4, 5, 6$): $\neg MEO1 \wedge MEO2 \wedge$ the (k-1)th subclause of ($I6$) with $i = 2$.

3) Proof of Invariants: The invariants ($I2$) and ($I7$) are the same as the assertions (OR3) and ($ME3$). The other invariants follow from the following analysis, which is based on case analysis. For each of the above classes of system states, we consider different values for the circuits of the arbiter implementation for the next system state. The proof rule (Proof 3) for the **while** operator and the assumptions of the arbiter specification of Section III-A2 are used to limit the number of possible cases. This leads to an exploration of all system states that are reachable from an initial state. The approach is a kind of "reachability analysis," and is related to symbolic execution.

Clearly, the invariants hold in the initial state (all circuits have the value zero). We note that the initial state satisfies the assertion (State 1). As long as the system is in "state" 1, assertions (ORi) imply that $MEIi = URi$. [Here and in the following, we write casually "state" x to indicate a state that satisfies (State x)]. We now consider an arbitrary transition to a next state s' from a present state satisfying (State 1). The following cases are possible for the next state s' ($i = 1, 2$).

Case 1: Assume $\neg MEOi' \Rightarrow_1 \neg TRi' \Rightarrow_2 \neg TAi' \wedge \neg UAi' \Rightarrow 3 \neg SRi'$.

Case 2: Assume $MEOi'$.

Case 2.1: Assume $\neg URi' \Rightarrow_4 UAi' \wedge \neg SRi' \Rightarrow 5 \neg SR' \Rightarrow_6 \neg SA' \Rightarrow_7 \neg UAi'$ (contradiction).

Case 2.2: Assume $URi' \Rightarrow TRi'$.

Case 2.2.1: Assume $TAi' \Rightarrow SRi' \Rightarrow SR'$.

Case 2.2.1.1: Assume $SA' \Rightarrow UAi'$.

Case 2.2.1.2: Assume $\neg SA' \Rightarrow \neg UAi'$.

Case 2.2.2: Assume $\neg TAi' \Rightarrow \neg SRi' \Rightarrow \neg SR' \Rightarrow \neg SA' \Rightarrow \neg UA'$.

In order to make the reasoning behind the above cases more clear, we have indicated for the first of the above implications an index that refers to the following explanations.

1) This follows from (ANDi).

2) Using the proof rule (Proof 3) explained in Section II-B for ($TiFC3$) yields

$$\neg TAi \wedge \neg TRi \wedge \neg TRi' \Rightarrow \neg TAi'$$

which implies $\neg TAi'$ since $\neg TAi \wedge \neg TRi$. Similarly, the rule applied to ($Ci2$) yields

$$\neg UAi \wedge (\neg TRi \vee \neg SA) \wedge (\neg TRi' \vee \neg SA') \Rightarrow \neg UAi'$$

which implies $\neg UAi'$.

3) This follows from ($Ck2$) and the fact that $\neg TAi$ holds. The proof is similar as for 2).

4) ($ME1$) implies that $MEIi'$ is true; therefore (ORi) implies UAi'. The second part follows from (ANDi).

5) This follows from ($Ck2$) similarly as in 2).

6) This follows from ($SFC3$) similarly as the first part of 2).

7) This follows from ($Ci2$).

It is easy to check that all cases (except Case 2.2 which cannot occur) satisfy the invariants. We note that the Case 1 (for $i = 1$ and 2), 2.2.1.1, 2.2.1.2, and 2.2.2 (for $i = 1$) satisfy the assertions (State 1), (State 2), (State 3), and (State 4), respectively. This means that the next state may still satisfy (State 1), or transitions may occur to the "states" 2, 3, and 4. Similarly, transitions may occur to "states" 2′, 3′, and 4′.

In a similar manner one can determine the possible transitions from other "states." The result is shown in Fig. 5, where an arrow from "state" x to "state" y indicates that for a present state satisfying (State x), there is a possible next state satisfying (State y), and all such possibilities are indicated. We do not give the details for deriving Fig. 5, but consider in the following only some cases that are of particular interest. Starting in a state satisfying (State 2), and assuming the case that $MEOI' \wedge TAi'$ is true, we have the following subcases.

Case 1: Assume $URi' \Rightarrow SRi' \wedge TRi' \Rightarrow SR'$.

Case 1.1: Assume $SA' \Rightarrow UAi'$.

Case 1.2: Assume $\neg SA' \Rightarrow \neg UAi'$.

Case 2: Assume $\neg URi' \Rightarrow \neg TRi' \wedge \neg SRI' \Rightarrow \neg SR' \Rightarrow SA' \Rightarrow \neg UAi' \Rightarrow \neg MEIi' \Rightarrow \neg MEOi'$ (contradiction).

We note that the four-cycle signaling scheme, as defined in Section III-A1, allows that the response of the acknowledge circuit is given simultaneously with the setting or resetting of the request circuit. This is realized in Case 1.1 above. It also allows that the resetting of the request circuit or the next request is given simultaneously with the response of the acknowledge circuit to the last value change of the request circuit. We see from the above consideration of cases that starting in "state" 2, the next "state" may set the user acknowledge circuit (Case 1.1), but it is not possible that the user simultaneously resets its request (Case 2). This is a restriction on the behavior of the user module that is not implied by the four-cycle signaling convention specified in Section III-A1.

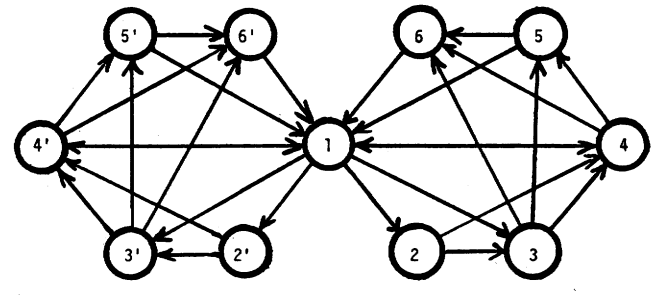


Fig. 5. Complete transition diagram for the arbiter implementation of Fig. 3.

Another interesting example are the transitions from "state" 6. Starting in a state satisfying (State 6), we have the following possibilities.

Case 1: Assume $SA' \Rightarrow UAi' \Rightarrow \neg URi' \wedge MEOi' \Rightarrow \neg TRi' \Rightarrow \neg TAi' \Rightarrow \neg SRi' \Rightarrow \neg SR'$.

Case 2: Assume $\neg SA' \Rightarrow \neg UAi'$.

Case 2.1: Assume $\neg URi' \Rightarrow \neg MEIi' \Rightarrow \neg MEOi' \Rightarrow \neg TRi' \Rightarrow \neg TAi' \Rightarrow \neg SRi' \Rightarrow \neg SR'$.

Case 2.2: Assume $URi' \Rightarrow MEIi' \Rightarrow MEOi' \Rightarrow TRi' \Rightarrow UAi'$ (contradiction).

This shows that the next state either satisfies (State 6) (Case 1, there is no progress) or (State 1) (Case 2.1). Therefore, the system will always come back to "state" 1, i.e., this state cannot be skipped, as "states" 2 and 3 may be, for example. This observation is important for the consideration of liveness below.

The contradiction in Case 2.2 shows that a user module may not make a new request at the same time as the acknowledge circuit is reset. This is another restriction on the behavior of the user module that is not implied by the four-cycle specification of Section III-A1. This stronger assumption on the user modules may be expressed by adding to the assumptions of the arbiter specification in Section III-A2 the assertion

($UiCF6$) $(UAi \neq x) \wedge (\textbf{next}\ UAi = x) \Rightarrow (\textbf{next}\ URi = x)$.

4) Proof of the Arbiter Assertions: We can now use the invariants to derive the safeness assertions of the arbiter. The assertions ($UiFC3$), ($TiFC1$), and ($SFC1$) can be verified by observing that the proof rule for the **while** operator is satisfied for all possible transitions. As shown above, however, it is necessary to make the stronger assumption ($UiCF6$) about the behavior of the user modules.

The mutual exclusion property ($A1$) follows easily from ($I7$) and ($I4$). Also, the assertions ($A2$) and ($A3$) can be easily derived from the invariants.

The arbiter liveness assertions ($UiFC4$), ($UiFC5$), ($TiFC2$), ($SFC2$), and ($A4$) are verified by showing that as long as user

requests are present, certain transitions must occur that eventually lead back to the initial state. This approach is similar to the one described in [7]. Assume, for example, that the present state satisfies (State 1). If we use the following assertion which is proven below

$$(ME \text{ fairness}) \qquad \text{State } 1 \wedge URi \Rightarrow \nabla\, MEOi \ (i = 1, 2)$$

then the consideration of possible cases in Section III-D3 implies the assertion

$$\text{State } 1 \wedge UR1 \Rightarrow \nabla\, \text{State } 2 \vee \text{State } 3 \vee \text{State } 4).$$

Similarly, if the system state satisfies (State 2), assumption ($TIFC4$) proves that

$$\text{State } 2 \Rightarrow \nabla\, TA1$$

which implies by consideration of the different cases that

$$\text{State } 2 \Rightarrow \nabla\, (\text{State } 3 \vee \text{State } 4).$$

We conclude that the liveness assumptions of the surrounding modules, the (ME fairness) assertion above, and the considerations of possible transitions of the arbiter implementation prove that when a user request is active, the system will perform eventually a certain number of transitions that lead through different states and back to "state" 1, as shown in Fig. 5. This implies the liveness assertions of the arbiter. Assertions ($A4$) and ($A5$), in particular, follow from the observation that the transition sequence leads through "state" 3 and/or 4, for which $URi \wedge TRi \wedge SR$ is true.

The (ME fairness) assertion can be proven as follows. Let us do the considerations for $i = 1$. First, we show that $\nabla \Box \neg MEO2$ holds. This is true since whenever $MEO2$ holds a sequence of transitions will lead back to a state satisfying (State 1), for which $\neg MEO2$ holds. Now we assume that (State 1 $\wedge$ $UR1$) holds in the present state, and $\neg \nabla MEO1$, which implies $\Box \neg MEO1$. (The temporal logic axioms used are explained in [1].) This further implies $\Rightarrow \Box \neg TR1 \Rightarrow \neg \Box UA1 \Rightarrow \Box UR1 \Rightarrow \Box MEI1$. Then we can use the property ($ME4$) to show that $\nabla MEO1$ holds, which is a contradiction to the assumption. Therefore, $\nabla MEO1$ must hold. We note that it is essential for this proof that the system goes back to "state" 1 before servicing another request. This is related to the fact that a user module must wait at least to the next instant after the acknowledge circuit is reset before it may make the next request (see above).

IV. Comments and Conclusions

We conclude this paper by discussing some particular points and giving some concluding remarks. As may be expected, we found that temporal logic was a versatile tool for specifying various properties of hardware modules. We hope that the example of this paper has also convinced the reader of the suitability of temporal logic for the specification of asynchronous systems. As discussed in Section II-C, traditional logic is useful for specifying combinational circuits and certain safeness properties of processes, the temporal **while** operator allows the specification of memory properties and other safeness properties of processes, while the "eventually" and "henceforth" operators are useful for the specification of the liveness properties of processes.

Section III gives the example of a verification of an arbiter implementation. We note that this verification is basically a "reachability analysis" where assertions are used to characterize sets of possible system states. Using the proof rules for the **while** operator given in Section II-B, the verification of safeness is relatively straightforward. It involves mainly the consideration of a large number of different cases. An automated proof system can be useful here.

The proof of liveness and fair scheduling between the two users involves fewer cases, but is logically more complex than the safeness proof. Here the power of the temporal operators $\Box$ and ∇ are particularly useful.

It is interesting to note that the specification method used in this paper does not make any distinction between input and output. For the specification of a given module in a system, the possible values for input and output circuits are mentioned, but nothing is said about whether a given value is determined by the module in question (output), or received from another connected module (input). This does not mean that the distinction between input and output is not an important one. For instance, the verification proofs considering the different cases in the Sections III-C and III-D3 are guided by the flow of information from outputs to inputs of submodules, corresponding to the intuition of the author. Clearly, the distinction is also crucial for the hardware realization of the submodules.

While the design error in an arbiter implementation mentioned in Section III-C was found during the informal reachability analysis described in that section, the more formal proof described in Section III-D uncovers the following more subtle possible timing problem. It was shown in Section III-D3 that a user submodule Ui must have observed the acknowledge circuit down at least one instant in time before it makes the next request. In our computational model of discrete time, what does this requirement really mean? We assume that all value changes of one transition have effectively been propagated by the hardware realization before the next transition may occur. In the case of the user interaction considered here, it means that the change of the acknowledge circuit to the value zero must have been propagated from the input SA through the C-element Ci, the OR-gate ORi, and through the output circuit UAi to the user module before the latter sets the input circuit URi to one. If this assumption is not satisfied, a "timing problem" may occur. For example, in the case that the user module is fast (responds with the next request with a short delay) compared to the speed with which the value change of the acknowledge propagates through the OR-gate ORi, the ORi output $MEIi$ may stay up. This implies that the corresponding output of the ME-element remains up, and fair scheduling is precluded. We note that this difficulty is not present in Seitz' arbiter implementation (see [4] and Section III-C).

Acknowledgment

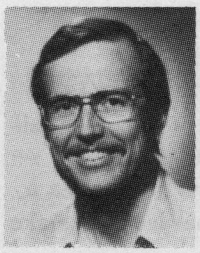
The author would like to thank J. Clark for suggesting the example of the arbiter, and S. Owicki for many useful comments and suggestions on earlier versions of this paper.

References

[1] Z. Manna and A. Pnueli, "The modal logic of programs," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Rep. STAN CS 79751, 1979.

[2] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980, ch. 7.

[3] G. V. Bochmann, "High-level modular hardware design and interfaces," Dep. Inform. Recherche Oper., Univ. of Montreal, Montreal, P.Q., Canada, Pub. 393.
[4] C. L. Seitz, "Ideas about arbiters," *LAMBDA*, pp. 10–14, First Quarter 1980.
[5] R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 371–384, July 1976.
[6] G. V. Bochmann, "Architecture of distributed computer systems," *Lecture Notes in Computer Science No. 77*. Heidelberg: Springer, 1979, ch. 4.
[7] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," Tech. Rep., Oct. 1980.



Gregor V. Bochmann received the Diploma in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. He is currently Associate Professor in the Departement d'Informatique et de Recherche Operationnelle, Université de Montreal. His present work is aimed at design methods for communication protocols and distributed systems. From 1977 to 1978 he was a Visiting Professor at the Ecole Polytechnique Fédérale, Lausanne, Switzerland. From 1979 to 1980 he was a Visiting Professor in the Computer Systems Laboratory, Stanford University, Stanford, CA.

A Data Structure for Parallel L/U Decomposition

JOCHEN A. G. JESS AND H. G. M. KEES

***Abstract*—Some new results are presented concerning the pivoting of large systems of linear equations with respect to parallel processing techniques. It will be assumed that the processing of a pivot takes one time slot. The pivoting problem is studied by means of an associated graph model. Given a triangulated graph a set of label classes is established. Class k contains all pivots which may be processed in parallel during the kth time slot. The label classes are used to establish the elimination-tree (e-tree). The e-tree is a spanning tree for the given graph. The critical path in the e-tree indicates the minimum number of time slots necessary to complete the L/U-decomposition. Furthermore, the earliest and latest admissible time slot for the processing of every pivot may be derived, such that the critical path is not affected. The e-tree can be seen as a data structure to guide parallel processing based on sparsity.**

***Index Terms*—Elimination-tree, L/U decomposition, parallel processing, schedule, sparse matrix pivoting, task graph, tearing, triangulated graph.**

I. Introduction

THE parallel solution of the linear equation system $Ax = b$ by L/U decomposition is investigated. The L/U decomposition offers basically two different sources of parallelism.

- Given some pivot, the products involving coefficients from the pivot-row and pivot-column can all be computed in parallel in all L/U decomposition schemes. Among others, Pottle and Wong [1] have proposed a computing unit, which exploits this parallelism partially.
- Sparsity of A implies that the computations associated with certain pivots can be executed simultaneously. This aspect has been studied by various authors [2], [3]. In fact, methods [4]–[8] known as "tearing" and the decomposition into "bordered block diagonal form" or "bordered block triangular form" are in close relation to this issue.

Manuscript received February 16, 1981; revised October 1981.
The authors are with the Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands.

At this point Wing and Huang's [9] approach is mentioned which attempts to consider both sources in one theoretical model.

A new model will be offered here. Central to the model is the "elimination-tree" ("e-tree"). The elimination-tree is derived from a graph model of a structurally symmetric matrix [10]. The algorithm generating the e-tree is simple. Furthermore, the e-tree provides a simple and rigorous guide for the organization of store and programflow for the L/U decomposition. The e-tree is mainly used to account for parallelism due to sparsity. The inherent parallelism of L/U decomposition is accounted for by assuming that whatever work is involved by processing one pivot can be done in a fixed "time-slot."

Section II starts with some necessary graph notations. Section III constitutes a somewhat elaborate introduction into the background of the theory. Generation and theory of the e-tree are presented in Section IV. Section V illustrates the application of the concept.

II. Graph Notations

A "graph" $G = (V, E)$ is defined by a set V of elements called "vertices" and a set E of unordered distinct vertex pairs called "edges"; thus

$$E \subseteq \{(u, v) \mid u, v \in V; u \neq v\}.$$

If $(u, v) \in E$, u and v are "adjacent" vertices and the edge (u, v) is "incident" to u and v.

The set "inc(v, E)" denotes the set of edges incident to vertex v. Thus,

$$\text{inc}(v, E) := \{(u, v) \in E \mid u \in V\}.$$

The set "adj(v, E)" denotes the set of vertices adjacent to vertex v. Thus,

$$\text{adj}(v, E) := \{u \in V \mid (u, v) \in E\}.$$

A set $W \subseteq V$ identifies a "subgraph" $G(W) = (W, E(W))$

0018-9340/82/0300-0231$00.75 © 1982 IEEE