The Design and Verification of Finite State Hardware Controllers

E.M.Clarke, S.Bose, M.C.Browne, O.Grumberg
July 1987
CMU-CS-87-145

This research was partially supported by NSF Grant MCS-82-16706. O. Grumberg, is currently on leave from Technion, Haifa and is partially supported by a Weizmann postdoctoral fellowship.
Table of Contents

1. Introduction 1
2. CTI. and EMC 2
3. Using SMI. to develop a DMA controller 4
4. Verifying the DMA controller 10
5. Directions for Future Research 15
List of Figures

Figure 3-1: DMA System 5
Figure 3-2: DMA System 6
Figure 3-3: CPU Interface 8
Figure 3-4: Address Comparator 9
Figure 3-5: Memory Interface 9
Figure 3-6: DMA Peripheral Interface 10
Figure 3-7: DMA Controller 11
Figure 4-1: Moore Machine For DMA Controller 12
Figure 4-2: The Model Checker Finds An Error 13
Figure 4-3: Counterexample Facility 14
Figure 4-4: Corrected Dma Controller 15
1. Introduction

Because finite state machines are such common components of VLSI circuits, a number of different state machine description languages (AMAZE, CUPL, SLIM, etc.—see [9] for a survey) have been devised. In general, these languages represent state machines at a very low level; most even require an explicit description of the state-transition behavior of the machine that is to be implemented. If the number of states is large, this can be a tedious and error-prone process. Moreover, a large state transition table developed by one designer may be difficult for another designer on the same project to modify or enhance. We have designed a programming language called SML (State Machine Language) that provides a succinct notation for specifying complicated finite state machines [2]. In addition, we have developed an automatic temporal logic verifier that can be used to make sure that SML programs are correct. This paper illustrates the power of our approach by showing how these two tools can be used to debug a fairly complicated DMA controller.

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. Our language, unlike the ones mentioned above, has many of the standard control structures found in modern high-level imperative programming languages, including a while statement, a conditional, a case statement, and a parallel execution statement. There is even a simple mechanism for declaring non-recursive procedures. However, the only data types that we allow are booleans and fixed width integers. Consequently, any program written in SML has only a finite number of states. SML programs are compiled into state transition tables which can then be implemented in hardware as PALs, PLAs, or ROMs. A post-processor is available that converts the state tables produced by the SML compiler in a format which is compatible with the Berkeley VLSI design tools. The language and its compiler have been used successfully within our department to develop a number of different hardware controllers.

Although there has been some work on the use of high level languages for describing state machines ([6], [7]) our system is unique in that the state transition table produced by the SML compiler can also be given to a temporal logic verifier that allows certain properties of the state machine to be verified automatically. Temporal logic is a formal system for reasoning about the occurrence of events in time without introducing time explicitly. The variant of temporal logic that we use for specification is a propositional, branching-time temporal logic called CTL or Computation Tree Logic ([4], [5]). Typical operators include AG f, which holds in the present state provided that f holds globally along all possible computations paths starting from the present state, and AF f, which holds in the present state provided that f inevitably holds in the future in all possible computations. These operators permit complicated timing properties to be expressed as formulas in the logic. For example, in our logic it is easy to express the property that if some event E_1 occurs then another event E_2 must inevitably follow: AG(E_1 → AF E_2).

This research was partially supported by NSF Grant MCS-82-16706. O. Grumberg is currently on leave from Technion, Haifa and is partially supported by a Weizman postdoctoral fellowship.
We call our verifier a model checker ([1], [5]). It uses an algorithm somewhat similar to those found in the information propagation phase of optimizing compilers to determine the truth or falsity of temporal logic formulas. It determines whether a temporal logic formula is true or not by traversing the state graph of the Moore machine and searching for a counterexample. The program will always answer true or false and is guaranteed to find a counterexample if there is one. The complexity of the algorithm is linear in the number of states of the Moore Machine, but exponential in the number of inputs and outputs. In practice the worst case complexity is seldom observed. In fact, our verifier averages approximately 100 states per second! The counterexample trace produced when a formula is not true is quite useful for debugging SML programs [3].

Our paper is organized as follows: In Section 2 we describe the version of temporal logic that we use as a specification language. In Section 3 we illustrate how SML might be used by writing a DMA controller. In Section 4 we show how our temporal logic verifier can be used to debug the DMA controller. The paper concludes in Section 5 with a discussion of some possible language extensions.

2. CTL and EMC

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL (computational tree logic). This logic is essentially the same as that described in [5]. The syntax for CTL is as follows: Let $\mathcal{P}$ be the set of all atomic propositions in the language $\mathcal{L}$, then:

1. Every atomic proposition $P$ in $\mathcal{P}$ is a formula in CTL.

2. If $f_1$ and $f_2$ are CTL formulas, then so are $\neg f_1$, $f_1 \land f_2$, $\text{AX} f_1$, $\text{EX} f_1$, $\text{A}[f_1 \cup f_2]$ and $\text{E}[f_1 \cup f_2]$.

In this logic the propositional connectives $\neg$ and $\land$ have their usual meanings of negation and conjunction. $\text{A}$ and $\text{E}$ are path quantifiers—$\text{A}$ means "for every computation path" and $\text{E}$ means "for some computation path." The temporal operator $\text{X}$ is the next time operator. Hence, the intuitive meaning of $\text{AX} f_1$ ($\text{EX} f_1$) is that $f_1$ holds in every (in some) immediate successor state of the current state. The temporal operator $\text{U}$ is the strong until operator. The intuitive meaning of $\text{A}[f_1 \cup f_2]$ ($\text{E}[f_1 \cup f_2]$) is that for every computation path (for some computation path), there exists an initial prefix of the path such that $f_2$ holds at the last state of the prefix and $f_1$ holds at all other states along the prefix.

We also use the following syntactic abbreviations:

- $f_1 \lor f_2 \equiv \neg (\neg f_1 \land \neg f_2)$. $f_1 \rightarrow f_2 \equiv \neg f_1 \lor f_2$ and $f_1 \Rightarrow f_2 \equiv (f_1 \rightarrow f_2) \land (f_2 \rightarrow f_1)$.
- $\text{AF} f_1 \equiv \text{A}[\text{true} \cup f_1]$ which means for every path, there exists a state on the path at which $f_1$ holds.
- $\text{EF} f_1 \equiv \text{E}[\text{true} \cup f_1]$ which means that for some path, there exists a state on the path at which $f_1$ holds.
- $\text{AG} f_1 \equiv \neg \text{EF} \neg f_1$ which means for every path, at every node on the path $f_1$ holds.
- $\text{EG} f_1 \equiv \neg \text{AF} \neg f_1$ which means for some path, at every node on the path $f_1$ holds.
We also define the weak until operator \( \mathcal{u} \) which is similar to the strong until except that it does not imply that the second condition is inevitable. For example, \( A[f_1 \mathcal{u} f_2] \) is satisfied when all paths have an initial sequence of states satisfying \( f_1 \) immediately followed by a state satisfying \( f_2 \) or consists of an infinite sequence of states satisfying \( f_1 \). The weak until can be defined by syntactic abbreviation \( A[f_1 \mathcal{u} f_2] \equiv \neg E[\neg f_1 \cup (\neg f_1 \land \neg f_2)] \) which means that for every computation path, \( f_1 \) is true in all states preceding the (first) state in which \( f_2 \) is true.

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

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

A path is an infinite sequence of states \( s_0, s_1, s_2, \ldots \) such that for every \( i \), \(<s_i, s_{i+1}> \in R \). For any structure \( \mathcal{M} = (S, R, \Pi) \) and state \( s_0 \in S \), there is an infinite computation tree with root labeled \( s_0 \) such that \( s \rightarrow t \) is an arc in the tree iff \(<s, t> \in R \).

The truth in the structure is expressed by \( \mathcal{M}, s_0 \models f \), meaning the temporal formula \( f \) is satisfied in the structure \( \mathcal{M} \) at state \( s_0 \). The semantics of the temporal formulas are defined inductively as follows.

1. \( s_0 \models P \) \( \Rightarrow \) \( P \in \Pi(s_0) \).
2. \( s_0 \models \neg f_1 \) \( \Rightarrow \) \( s_0 \not\models f_1 \).
3. \( s_0 \models f_1 \land f_2 \) \( \Rightarrow \) \( s_0 \models f_1 \) and \( s_0 \models f_2 \).
4. \( s_0 \models A f_1 \) \( \Rightarrow \) for all states \( t \) such that \(<s_0, t> \in R \), \( t \models f_1 \).
5. \( s_0 \models E f_1 \) \( \Rightarrow \) for some state \( t \) such that \(<s_0, t> \in R \), \( t \models f_1 \).
6. \( s_0 \models A [f_1 \cup f_2] \) \( \Rightarrow \) for all paths \( (s_0, s_1, s_2, \ldots) \) there exists a \( k \geq 0 \) such that \( s_k \models f_2 \) and for all \( 0 \leq j < k \), \( s_j \models f_1 \).
7. \( s_0 \models E [f_1 \cup f_2] \) \( \Rightarrow \) for some path \( (s_0, s_1, s_2, \ldots) \) there exists a \( k \geq 0 \) such that \( s_k \models f_2 \) and for all \( 0 \leq j < k \), \( s_j \models f_1 \).

There is a program called EMC (extended model checker) that verifies the truth of a formula in a model using these definitions. It uses efficient graph-traversal algorithms to check a formula in time linear in the size of the graph and in the length of the formula. If the CTL structure is represented as a Moore machine, the complexity remains linear in the length of the formula and the number of states but is exponential in the number of inputs and outputs. (See [1] and [5] for details).

There are two additional features of the model checker that turn out to be particularly useful in practice. The first extension is the addition of fairness constraints. Occasionally, we are only interested in the correctness of fair execution sequences. For example, we may wish to consider only execution sequences in which
some process that is continuously enabled will eventually execute. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of CTL slightly. Initially, the model checker will prompt the user for a series of fairness constraints. Each constraint can be an arbitrary formula of the logic. A path is said to be fair with respect to a set of fairness constraints if each constraint holds infinitely often along the path. The path quantifiers in CTL formulas are now restricted to fair paths. Examples of fairness constraints can be found in Section 4. In [1] and [5] we show that handling fairness in this manner does not change the linear time complexity of the model checker.

The second feature is a counterexample facility. When the model checker determines that a formula is false, it will attempt to find a path in the graph which demonstrates that the negation of the formula is true. For instance, if the formula has the form $\text{AG} \neg f$, our system will produce a path to a state in $\neg f$ holds. This feature is quite useful for debugging. EMC is written in C and runs on a VAX 11/780 under Unix.

3. Using SML to develop a DMA controller

We have developed a language called SML (state machine language) [2] for describing complicated finite state machines. A program written in SML is compiled into a Moore machine, which can then be verified using the model checker or implemented in hardware. 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. A system may be modeled by some concurrently executing processes, each of which may correspond to one of its components. The interaction between these components occurs with the help of signals which are represented as internal variables. These signals may be unidirectional (driven by a specific component) or bidirectional. The hardware implementation of boolean variables may be declared to be either active high or active low. 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. There is also a simple mechanism for declaring non-recursive procedures.

In this section we show how SML can be used to construct a simple DMA controller. In the next section we show how our model checking program can be used to guarantee that our DMA design meets certain timing properties expressed in the logic CTL. Direct memory access (DMA) is a technique that permits blocks of data to be transferred directly from an I/O peripheral to main memory without using any of the CPU’s data or address registers. With this technique (which is sometimes called cycle stealing) the DMA peripheral can execute a memory access at any time that the CPU is not using the memory. Moreover, the CPU will be able to continue with its normal operations until it reaches a point where it needs to make a memory access but a DMA operation is still in progress. Our design for a DMA controller is loosely based on one that is described in [8]. Although it is probably much simpler than most actual DMA systems, the relatively small number of
states in our SML program and the short length of time that it takes to verify the program (a few seconds per specification) mean that our tools should also be useful for much larger and more complicated designs as well.

Figure 3-2 shows the global structure of the SML program for the DMA system. In order to be able to prove interesting properties of the DMA controller in the next section, our design must represent the various system components that interact with the DMA controller. Thus, in our design, there are five different processes: one for the DMA controller, one for the DMA peripheral, one for CPU, one for main memory, and finally one for the address comparator that is used in determining whether block transfer is complete. These five processes together specify the single finite state machine representing the DMA system. The behavior of these processes will be explained in more detail below. Figure 3-1 shows how they are connected together. In the diagram boxes represent the five processes. Unidirectional signals are shown as unidirectional arrows. For signals which have more than one driver, bidirectional arrows have been used.

![DMA System Diagram]

**Figure 3-1: DMA System**
#define CPU true
#define DMA false
#define MEMREAD false
#define MEMWRITE true

program DmaSystem;

-- cpu declarations
input MemReq, ReqType;
internal CpuReq, TransferReq, DmaType;

-- comparator declarations
input ComparatorResult;
internal ComparatorSet;

-- memory declarations
input MemFinished;
internal Busy, Read, Write;

-- peripheral Declarations
input DeviceReady;
internal DmaAcc, DmaValid, DmaReq;

-- dma controller declarations
internal ActivateComparator, DmaDone, DmaCont;
internal DmaEnd, MemGrant;

procedure wait(exp)
    while !(exp) do loop skip endloop
endproc

parallel
loop ... endloop || -- cpu interface
loop ... endloop || -- address comparator
loop ... endloop || -- memory interface
loop ... endloop || -- dma peripheral
loop ... endloop -- dma controller
endparallel
endprog

Figure 3-2: DMA System
The CPU interface controls the various types of memory operations that involve the CPU. It consists of a case statement with four alternatives that is repeatedly executed. The first two alternatives handle memory read and write requests generated directly by the CPU. The third alternative is executed to initiate a transfer involving the DMA peripheral. The default case is simply a skip statement which will be continually selected while the CPU is executing an operation that does not involve main memory. The first two alternatives have essentially the same steps. *CpuReq* is raised to indicate that a memory access is needed. The DMA controller will raise *MemGrant* in response, if the memory is not needed for a DMA transfer. The CPU will provide the appropriate inputs for the memory operation and raise *Read or Write*. The memory interface will raise *Busy* to indicate that it has begun to process the request. At this point the *Read or Write* signal may be lowered. The CPU will wait until the memory operation has been completed (i.e. until *Busy* goes low) and then transfer any results of the memory operation to the necessary CPU registers. *CpuReq* can then be lowered as well. In the third alternative *DmaType* is set to *false* for *true* depending on whether the DMA request is a read or write operation. Then *TransferReq* is raised to start the DMA peripheral.

The address comparator determines whether a DMA transfer has finished by comparing the contents of the DMA address register (*DmaAddr*) with the contents of the DMA last word register (*DmaLwr*). Initially, *DmaAddr* and *DmaLwr* are loaded with the addresses of the first and last words in the block to be transferred between main memory and the DMA peripheral. *DmaAddr* is incremented each time the transfer of another word in the block has been completed. When the *ActivateComparator* signal becomes high, the two registers are compared. *DmaDone* is assigned the value *true* if the two registers have the same value and *false* otherwise. The signal *ComparatorSet* is then raised to indicate that the comparison is complete. *ComparatorSet* is lowered after *DmaReq* becomes low.

The memory interface is also quite simple. *MemAdr* will contain the address of a word in memory. A read operation will transfer the contents of that word to *DataOut*. A write operation, on the other hand, will replace the contents of the addressed memory location by the value on *DataIn*. A read (or write) operation is initiated by raising *Read* (or *Write*) when *Busy* is low. The memory will respond by latching *MemAdr* (and *DataIn* in the case of a write operation) and raising *Busy*. *Busy* will be lowered again when *MemFinished* becomes high, indicating that the transfer has been completed.

The DMA peripheral will wait until the CPU initiates a DMA transfer by raising *TransferReq*. When *DeviceReady* becomes high, it will raise *DmaReq* to indicate that it is ready to transfer another word between the device (itself) and main memory. If the DMA request is a memory write, then the peripheral will transfer the data to be written to the *DmaBus* and wait for *DmaAcc* to be asserted. If the request is a memory read, the peripheral will wait until *DmaValid* becomes high to transfer data from the *DmaBus* to its internal registers. Then, it will raise *DmaAcc* to inform the Dma Controller that it has finished. If the entire DMA transfer has been completed (i.e. *DmaEnd is true*), the peripheral will lower *DmaReq*, return to the top of the outer loop, and wait for *TransferReq* to become high again. If the the transfer has not yet been completed and must be
loop
  switch
  case (MemReq == CPU) & (ReqType == MEMREAD):
    raise(CpuReq);
    wait(MemGrant);
    --transfer cpu generated address to MemAdr
    raise(Read);
    wait(Busy);
    lower(Read);
    wait(!Busy);
    --transfer memory output to cpu register
    lower(CpuReq);
    break;
  case (MemReq == CPU) & (ReqType == MEMWRITE):
    raise(CpuReq);
    wait(MemGrant);
    --transfer cpu generated address to MemAdr
    --also transfer cpu data to DataIn
    raise(Write);
    wait(Busy);
    parallel
      lower(Write) || lower(CpuReq)
    endparallel;
    wait(!Busy);
    break;
  case (MemReq == DMA) & !TransferReq:
    DmaType := ReqType;
    --Initialize DmaAdr and DmaLwr
    raise(TransferReq);
    break;
  default: skip;
  endswitch
endloop

Figure 3-3: CPU Interface

continued (i.e. DmaCont is true), the peripheral will lower DmaReq and return to the top of the inner loop to wait until the DMA peripheral is ready again.

The DMA controller coordinates the actions of the CPU, the main memory, and the DMA peripheral. If a DMA request occurs when main memory is not busy, it will arrange for the DmaAdr to be incremented and for the comparator to be activated. If the DMA request is a memory write, the controller will cause the contents of the DmaAdr to be transferred to the MemAdr and also for the data on the DmaBus to be transferred to the memory's DataIn register. It then raises the Write signal to initiate the memory operation and waits for Busy to go high. When the memory operation has been started, it raises DmaAcc to inform the
Figure 3-4: Address Comparator

Figure 3-5: Memory Interface

DMA peripheral that the DmaBus can be cleared. If the DMA request is a memory read, the controller will transfer the contents of the DmaAddr to the MemAddr and initiate the memory operation as in the previous case. When the operation has completed, it will transfer the data from the memory’s DataOut register to the DmaBus, assert DataValid, and then wait until the peripheral acknowledges by raising DmaAcc. When the comparator has finished comparing the values in the DmaAddr and the DmaLwr, the controller will set DmaEnd and DmaCont appropriately and wait until ComparatorSet goes low, indicating that the current phase in the DMA transfer has been finished.

Alternatively, if a CPU request occurs when the main memory is not busy, the controller will raise
loop
    wait(TransferReq);
loop
    wait(DeviceReady);
    if DmaType == MEMWRITE then
        --Transfer data to DmaBus
        raise(DmaReq);
        wait(DmaAcc);
        --clear DmaBus...
    else
        raise(DmaReq);
        wait(DmaValid);
        --Transfer data from DmaBus to
        --internal registers
        raise(DmaAcc);
        wait(!DmaValid);
        lower(DmaAcc)
    endif;
    wait(DmaCont | DmaEnd);
    if DmaEnd then
        lower(TransferReq); lower(DmaReq);
        exit endif;
    lower(DmaReq)
endloop
endloop

Figure 3-6: DMA Peripheral Interface

MemGrant to inform the CPU that it can proceed with its memory operation. If the memory is busy or if neither a DMA transfer nor a CPU memory access is needed, the DMA controller will simply loop. Note that because of the ordering of the two alternatives, a DMA transfer will always have priority over a CPU memory transfer. This is desirable, since data might be lost if service to the DMA peripheral were delayed too long a time.

4. Verifying the DMA controller

When the entire SML program is compiled, a deterministic Moore machine in FIF format (FSM Intermediate Format) is obtained. A small portion of the compiler output is shown in Figure 4-1. The Moore machine for the complete program has five inputs and fifteen outputs--internals are treated as outputs in FIF format. The ".H" suffix on each variable name indicates that all of the inputs and internals are active high. Before printing the FIF format, the compiler minimizes the Moore machine. In this case the minimized machine has 392 states and 922 transitions. Although this may seem like a large number of states, remember that the complete program has five major processes. Any one process (the DMA controller subprocess, for example) could be compiled separately and would have a much smaller number of states. The state numbers run from 0 to 391 and are prefixed by a "#" sign. The bit pattern on the same line with the state number tells which outputs are high in that state. Thus, all of the outputs are low in state 0, while Busy and DmaDone are high in the last state. In state 0 there are possible transitions to states 1, 2, 3, and 4. The transition to state 4 will occur if CpuReq and TransferReq are both high.
loop
  switch
    case DmaReq & !Busy:
      --increment DmaAddr
      raise(ActivateComparator);
      if DmaType == MEMWRITE then
        --transfer DmaAddr to MemAddr
        --and data on DmaBus to DataIn
        raise(Write);
        wait(Busy);
        parallel
          lower(Write)
        endparallel;
        raise(DmaAcc)
      endparallel;
      wait(!Busy);
      lower(DmaAcc);
    else
      --transfer DmaAddr to MemAddr
      raise(Read);
      wait(Busy);
      lower(Read);
      wait(!Busy);
      --transfer DataOut to DmaBus
      raise(DmaValid);
      wait(DmaAcc);
      lower(DmaValid)
    endif;
    wait(ComparatorSet);
    if DmaDone then
      raise(DmaEnd);
      wait(!ComparatorSet);
      lower(DmaEnd);
    else
      raise(DmaCont);
      wait(!ComparatorSet);
      lower(DmaCont);
    endif;
    lower(ActivateComparator);
    break;
  case CpuReq & !Busy:
    raise(MemGrant);
    wait(!CpuReq);
    lower(MemGrant);
    break;
  default: skip;
  endswitch
endloop

Figure 3-7: DMA Controller
NAME = DmaSystem;
STATES = 392;
CUBES = 922;
INPUTS = MemReq.H, ReqType.H, ComparatorResult.H,
MemFinished.H, DeviceReady.H;
MOORE-OUTPUTS = CpuReq.H, TransferReq.H, DmaType.H,
#0 0000000000000000
11XXX 4
10XXX 3
01XXX 2
00XXX 1

#391 000010000001000
XXX1X 388
XXX0X 391
#END

**Figure 4-1: Moore Machine For DMA Controller**

Figure 4-2 shows the transcript of a run of the model checking program described in Section 3 on our DMA protocol. For this example three fairness constraints are used. The first ensures that memory reads and writes always terminate. The second ensures that the DMA peripheral will always eventually be ready to transmit or receive data. The last is needed to guarantee that the CPU will always eventually get a chance to perform a memory operation.

The first specification asserts that if CpuReq becomes high, then eventually it will become low again. Therefore, if the CPU needs to perform a memory operation, it will eventually be able to do so. Time is measured in 1/60 of a second, so our program is able to determine that the first specification is satisfied in approximately 1.5 seconds! The second specification asserts that when TransferReq becomes high, eventually either DmaEnd or DmaCont will become high. The third assertion shows that it is impossible for ActivateComparator and MemGrant to be high at the same time. Since ActivateComparator is high while a memory read or write for the DMA is in progress, it follows that memory operations for the CPU and DMA are mutually exclusive.

For the next two specifications, we need the **weak until** operator **AW**. This operator is like the ordinary until operator **AU** except that its second argument is not required to eventually hold on every fair path. We define the weak until operator as a macro in terms of existential version of the ordinary until operator. The fourth and fifth specifications use the new operator to express the property that the type of a DMA transfer is not allowed to change while TransferReq is high.

The last three assertions do not have the correct truth values and, therefore, show that our program contains an error. The first two of these assertions state that DmaDone doesn’t change its value from the time that
ComparatorSet becomes high until either DmaCont or DmaEnd goes high. The last assertion shows that it is possible to get to a state in which the ActivateComparator can become high, but ComparatorSet is already true.

**CTL MODEL CHECKER (version B1.0)**

Reading DmaController...
Fairness constraint: ~Busy | MemFinished.
Fairness constraint: ~TransferReq | DeviceReady.
Fairness constraint: ~CpuReq | MemGrant.
Fairness constraint: .

|= AG(CpuReq -> AF ~CpuReq).
The formula is TRUE. time: 100

|= AG(~TransferReq -> AX(TransferReq -> AF(DmaEnd | DmaCont))).
The formula is TRUE. time: 186

|= EF(ActivateComparator & MemGrant).
The formula is FALSE. time: 43

|= AW(x,y) := ~E[~y U (~x & ~y)].
Macro AW defined.

|= AG(~TransferReq -> AX((TransferReq & DmaType) -> AW((TransferReq & DmaType), ~TransferReq))).
The formula is TRUE. time: 182

|= AG(~TransferReq -> AX((TransferReq & ~DmaType) -> AW((TransferReq & ~DmaType), ~TransferReq))).
The formula is TRUE. time: 194

|= AG((DmaDone & ComparatorSet) -> A[DmaDone U DmaEnd]).
The formula is FALSE. time: 309

|= AG((~DmaDone & ComparatorSet) -> A[~DmaDone U DmaCont]).
The formula is FALSE. time: 339

|= EF(~ActivateComparator &
    (EX ActivateComparator) & ComparatorSet).
The formula is TRUE. time: 133

**Figure 4-2:** The Model Checker Finds An Error
The model checker has a \texttt{trace} option that can be used to print an example execution for a true formula with an existential path quantifier or for a false formula with a universal path quantifier. This feature is important for debugging SML programs.

If we check the last assertion with this feature enabled, we obtain the execution shown in Figure 4-3. By examining this execution, it is relatively easy to see what is wrong with our program. In the DMA controller after we check the value of \texttt{DmaDone} we raise \texttt{DmaEnd} or \texttt{DmaCont} and wait for \texttt{ComparatorSet} to become low. Since \texttt{ActivateComparator} is not lowered until several statements later, it will still be high when \texttt{ComparatorSet} is lowered by the comparator process. The comparator process will incorrectly find that \texttt{ActivateComparator} is still true when it returns to the top of its loop. Thus, it will proceed to change \texttt{DmaDone} and reset \texttt{ComparatorSet}, causing the behavior that we have detected with the model checker.

\[|= \text{EF}(\neg \text{ActivateComparator} \& (\text{EX ActivateComparator} \\ & \& \text{ComparatorSet})].\]

The formula is TRUE.

Do you want to specify the input in the initial state? [n]

State 0-0:
State 1-0:
State 5-16: DeviceReady TransferReq
State 9-0: TransferReq DmaReq
State 21-0: TransferReq DmaReq ActivateComparator
State 37-0: TransferReq Read DmaReq
ActivateComparator
State 60-8: MemFinished TransferReq ComparatorSet
Busy Read DmaReq ActivateComparator
State 83-0: TransferReq ComparatorSet DmaReq
ActivateComparator
State 118-0: TransferReq ComparatorSet DmaValid
DmaReq ActivateComparator
State 138-0: TransferReq ComparatorSet DmaAcc
DmaValid DmaReq ActivateComparator
State 150-0: TransferReq ComparatorSet DmaAcc
DmaReq ActivateComparator
State 162-0: TransferReq ComparatorSet DmaReq
ActivateComparator DmaCont
State 178-0: TransferReq ComparatorSet
ActivateComparator DmaCont
State 200-4: ComparatorResult TransferReq
ActivateComparator DmaCont
State 222-16: DeviceReady TransferReq
ActivateComparator DmaCont
State 261-0: TransferReq ComparatorSet DmaReq
DmaDone
State 296-0: TransferReq ComparatorSet DmaReq
ActivateComparator DmaDone
time: 133

Figure 4-3: Counterexample Facility
The problem with the DMA controller can be fixed if we modify it as shown in Figure 4-4. In the new version the \textit{ActivateComparator} signal is lowered before \textit{DmaReq} is lowered. Thus, the comparator process will not be able to raise \textit{ComparatorSet} until \textit{ActivateComparator} is asserted again. When the modified program is compiled, a Moore machine with 272 states and 628 transitions is obtained. If the model checker is run on the new state transition graph, all of the CTL assertions in Figure 4-2 have the correct truth values. By checking additional properties in a similar manner, it is possible to obtain a high degree of confidence in the correctness of the program.

\begin{verbatim}
if DmaDone then
  parallel
    raise(DmaEnd)
  ||
  lower(ActivateComparator)
endparallel;
wait(!DmaReq);
lower(DmaEnd);
else
  parallel
    raise(DmaCont);
  ||
  lower(ActivateComparator)
endparallel;
wait(!DmaReq);
lower(DmaCont)
endif;
bend;
\end{verbatim}

\textbf{Figure 4-4: Corrected Dma Controller}

5. Directions for Future Research

In the process of designing and verifying the DMA controller, we realized a number of different ways in which the SML language and our temporal logic verifier could be extended in order to simplify this process. Some of these extensions are quite simple. For example, in writing CTL specifications it is frequently important to be able to assert that some property holds every time that control reaches a particular point in the program. This would be much easier if our language permitted statement labels—we could then simply write \texttt{AG (at <label> \rightarrow <property>)}. Since our language doesn’t have a \texttt{goto} statement, however, labels were omitted from the original language design.

Finally, some extensions involve more theoretical research. In order to check interesting properties of the DMA controller, we had to compile it with additional processes representing the various system components that interact with the controller. In order to implement the controller, however, we must compile the individual components separately and interconnect them with wires. If we find an error in the first approach, then most likely it will also be an error in the separately compiled version. However, if a specification checks in the first approach can we immediately assume that it will hold for the separately compiled version? The answer may depend on factors like the delays associated with wires and clearly needs more thought.
References


