SML - A High Level Language for the Design and Verification of Finite State Machines

M. C. Browne  
Carnegie-Mellon University

E. M. Clarke  
Carnegie-Mellon University

1. Introduction

Finite state machines are common components of VLSI circuits. Because they occur so frequently, many design tools have been written to layout finite state machines as PALs, PLAs, etc. Unfortunately, most of these tools require the user to enter the complete state transition table of the machine. If the number of states is large, this can be a difficult and error-prone process. Furthermore, large state transition tables are not easy for others to understand.

We have developed a programming language called SML (State Machine Language) that provides a succinct notation for describing complicated finite state machines. Our language 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 produces output which is compatible with the Berkeley VLSI design tools.

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 automatically verified. 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 ([4], [5]) contains a number of simple, easy-to-use operators that 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. We call our verifier a model checker ([2], [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. In practice, our verifier can check approximately 100 states per second. If a specification is not satisfied, the verifier produces a counterexample execution trace that shows why the formula is not true. The counterexample mechanism is quite useful for debugging SML programs [3].
Our paper is organized as follows: Section 2 contains a description of the SML language. In Section 3 two simple examples are presented that illustrate how the language can be used to construct hardware controllers. Section 4 contains a more complicated example (a bus controller) that illustrates how the temporal logic verifier may be used to debug SML programs. Section 5 briefly describes how the SML compiler works. In Section 6 SML is compared with other state machine description languages. The paper concludes in Section 7 with a discussion of some possible language extensions.

2. The SML Programming Language

An SML program represents a synchronous circuit that implements a Moore machine. At a clock transition, the program examines its input signals and changes its internal state and output signals accordingly. Since we are dealing with digital circuits where wires are either high or low, the major data type is boolean. Each boolean variable may be declared to be either an input changed only by the external world but visible to the program, an output changed only by the program but visible to the external world, or an internal changed and seen only by the program. The hardware implementation of boolean variables may also be declared to be either active high or active low. Internal non-negative 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 statement is provided to allow several statements to execute concurrently in lockstep.

An SML program has the following form:

```
program identifier;
   declaration list;
   statement list;
endprog
```

where identifier is the name of the program, declaration list is a sequence of declarations separated by semicolons, and statement list is a sequence of statements separated by semicolons.

2.1. SML Declarations

There are two types of declarations in SML: variable declarations and procedure declarations. Procedure declarations are of the form

```
procedure identifier (identifier list)
   statement list
endproc
```

where identifier is the name of the procedure and identifier list is a sequence of formal parameter identifiers separated by commas. SML uses call-by-name parameter passing, so that a procedure call of the form:
identifier (expression list)

has the same effect as statement list with the formal parameter identifiers replaced with the actual parameter expressions.

There are four different variable declarations in SML: internal integer, input boolean, output boolean, and internal boolean. The internal integer has the declaration has the form:

```
integer identifier[integer] initializer
```

where identifier is the variable being declared, integer is the number of bits used to implement the variable, and the initializer is either the empty string or " = integer". If the initializer is the empty string, the variable is initialized to zero. An integer variable can be thought of as an array of active high internal booleans. Therefore, it is possible to refer to an individual bit of an integer by using an array reference of the form:

```
identifier[integer]
```

where integer must be greater than or equal to zero and less than the number of bits in the variable. Identifier[0] is defined to be the least significant bit.

The input boolean declaration has the form:

```
input identifier type
```

where identifier is the variable being declared, and the type is either ".H" (active high), ".L" (active low), or the empty string (in which case the default is active high).

The output boolean and internal boolean declarations have the form:

```
output identifier type initializer
internal identifier type initializer
```

where identifier is the variable being declared, and the type is the same as for input booleans. A boolean initializer is either " = true", " = false", or the empty string (in which case the default value is false).

Several instances of the same type of variable declaration can be combined into one declaration by following the keyword (integer, input, output, or internal) with a list of the identifiers and other information separated by commas.

In order to clarify the use of variable declarations, consider the declaration list:

```
input A.H, B.L, C;
output D = true, E.L, F;
```

As a result of these declarations:

---

1SML procedures are actually implemented as macros.
• A and C are active high boolean inputs.

• B is an active low boolean input.

• D is an active high boolean output that will be high (active) in the initial state.

• E is an active low boolean output that will be high (inactive) in the initial state.

• F is an active high boolean output that will be low (inactive) in the initial state.

• X is an internal integer (that can have values from 0 to 7) that will have the value 5 in the initial state.

• Y is an internal integer (that can have values from 0 to 3) that will have value 3 in the initial state. (The binary representation of "7" needs three bits, but Y is only two bits long. Therefore, the value of the high order bit of "7" is lost.)

• Z is an internal integer (that can have values from 0 to 31) that will have the value 0 in the initial state.

2.2. SML Expressions

There are two types of expressions in SML, integer expressions and boolean expressions. An integer expression is either a natural number, an integer variable, or an application of an infix arithmetic operator to two integer expressions. The arithmetic operations in SML are sum ("+"), difference ("-"), product ("*"), quotient ("/"), or remainder ("%").

A boolean expression is either a boolean constant (true or false), a boolean variable (true if the variable is currently active), the negation of a boolean expression (prefix "!")", an application of an infix logical operator to two boolean expressions, or a comparison of two integer expressions. The logical operations in SML are conjunction ("&"), disjunction ("|"), equivalence ("=="), and exclusive or ("!="). The integer comparisons are equality ("=="), inequality ("!="), greater than (">"), or less than ("<").

In addition, int is a function that takes a boolean expression as a parameter and returns 1 if the expression is true and 0 if it is false. (Int can be used to convert a boolean expression into an integer expression.)

All binary operators associate from left to right. The operators have the following precedence (from lowest to highest):

=, /=, >, <

&

|

+, *

*, /, %
2.3. SML Statements

The semantics of SML programs are different from most programming languages, since we are not only interested in what a statement does, but also how much time the statement takes to execute. The basic idea in SML is that computation is instantaneous, but changing a variable takes one clock cycle. (We should note when we refer to the "time" that a statement takes we are referring to the execution time of the finite state machine. It is possible for computation to take no execution time since the computation is actually done at compile time.)

2.3.1. Sequencing of Statements

A statement may consist of two statements separated by a semicolon (";"). After the first statement has finished executing, the second one starts executing immediately.

2.3.2. Delay Statements

There are two methods of delaying execution:

- skip
- delay natural number

The skip statement will do nothing for one clock cycle. The delay statement will do nothing for natural number clock cycles. (delay 1 is identical to skip.)

2.3.3. Assignment Statements

Boolean input variables can not be assigned new values, since inputs are changed by the environment only. Boolean output and boolean internal variables may be changed by:

- raise (variable)
- lower (variable)
- invert (variable)

Each of these statements delays until the next clock transition, at which time the value of variable will be changed. The raise statement will assert variable (make it active), lower will deassert it, and invert will force a change of value. (Note that variable can also be an individual bit of an internal integer.)

Integer variables may be changed by:

- variable := integer expression

The integer expression is evaluated immediately, and after delaying until the next clock transition, variable will be assigned the low order bits of the two's complement representation of the expression's value.

2.3.4. Conditional Statements

There are two forms of conditional execution:

- if boolean expression then statement-1 else statement-2 endif
- if boolean expression then statement endif
In the first case, the boolean expression is evaluated. If the expression is true, statement-1 is executed, otherwise statement-2 is executed. Evaluating the expression and changing the flow of control does not take any time!

The second case is similar, except that nothing is done (in zero time!) if the boolean expression is false.

2.3.5. Looping Statements

There are two types of looping statements in SML: the while statement and the loop statement. The while statement has the form:

while boolean expression do loop statement endloop

At the beginning of the while, the boolean expression is evaluated, and nothing is done (in zero time) if the expression is false. If it is true, statement is executed. If statement completes execution in no time, the while statement delays until the next clock transition and then restarts the loop. If statement completes execution after some delay, the while statement is immediately restarted.

The loop statement has the form:

loop statement endloop

This statement is the same as while true do statement endwhile.

The exit statement has the form:

exit

The effect of this statement is to immediately jump out of the smallest enclosing while or loop statement. If there is no enclosing while or loop, this statement is an error.

2.3.6. The Switch Statement

The switch statement has the form:

switch
  case boolean expression-1: statement-1;
  case boolean expression-2: statement-2;
  ...
  default: statement-n;
endswitch

When the switch statement is entered, boolean expression-1 is evaluated. If the expression is true, statement-1 is executed, otherwise it is skipped (the evaluation and change in control flow takes no time, of course). After statement-1 is completed, boolean expression-2 is evaluated and statement-2 is executed if it is true and skipped if it is false. This procedure is continued until the default case is reached, whereupon statement-n is executed and the switch is completed. (The switch statement in SML is different from the C switch statement in that execution does not "fall through" cases!)
2.3.7. The Parallel Statement

The parallel statement provides a form of synchronous parallelism. This statement has the form:

```
parallel
    statement-1 ||
    statement-2 ||
    ...
endparallel
```

Each statement in the parallel examines the inputs and the current state and determines what changes it should be made to the output state at the next clock transition. The semantics of the parallel determine which of these changes are actually made. The rules are as follows:

1. If one or more of the statements executes an exit, the parallel does nothing and the exit causes a jump out of the smallest loop or while statement that encloses the parallel statement.

2. If one or more of the statements executes a break, the parallel does nothing and the break causes a jump to the statement following the parallel.

3. If one statement executes an exit and another statement executes a break, the statement closest to the beginning of the program is executed.

4. If none of the statements tries to change a variable, the variable remains unchanged.

5. If exactly one statement tries to change a variable, this change is made at the next clock transition.

6. If two or more statements try to change a boolean variable and they all agree on the new value, this change is made at the next clock transition.

7. If two or more statements try to change the same boolean variable and they do not agree on the new value, the variable remains unchanged.

8. Integer variables are treated as arrays of booleans for the purposes of finding their new values.

The parallel terminates when all of the statements in the parallel have finished executing or a break or exit is executed.

The break statement has the form:

```
break
```

The effect of this statement is to immediately jump out of the smallest enclosing switch or parallel statement. The main use of this statement is to prevent more than one case of a switch statement from executing. If there is no enclosing switch or parallel, this statement is an error.
One of the major uses of the break statement is to stop normal processing when some sort of "interrupt" occurs. For example, consider the following fragment:

```
loop
  parallel
  loop if RESET then break endif endloop
  ||
  --Normal processing
endparallel:  --Reset processing
endloop
```

In this fragment, normal processing is done until RESET goes high. When RESET goes high, the break statement jumps out of the loop AND the parallel to the reset routine. If SML had only one form of escape statement, it would be necessary to follow the loop with another escape in order to jump out of the parallel. However, we believe that the two forms of escape make this fragment easier to understand.

2.3.8. The Compress Statement

In some cases, the timing rules of SML prevent complicated relationships from being simply described without delaying for more than one clock cycle. For example, consider the following fragment from a blackjack dealing program.

```
program blackjack:
  inputs S1, S2, S3, SB:
  integer newcard[4] = 0:
  :
  newcard := 4 * SB + 2 * SB(31) + 4 * SB(32) + 8 * SB(33):
  if (newcard > 18 & newcard < 16) then
    newcard := 10
  endif

Figure 2-1: Card Decoding in a Blackjack Program
```

This fragment determines the value of a card presented to the input and stores the value in the integer newcard. Then, if the card is a face card (i.e. the number is between 11 and 13), the value of the card is 10. Unfortunately, the original assignment to newcard took one clock cycle and this new assignment takes another clock cycle. Although it is possible to avoid the original assignment to newcard by using the expression that is assigned to newcard instead of newcard in the if statement, this is very awkward. To alleviate this problem, SML has a compress statement of the form:

```
compress statement endcompress
```

The effect of the compress statement is calculated as if variable assignment takes no time in statement. Then, after delaying one clock cycle, the changes made by the compress statement actually take effect. (Even if the body of the compress does nothing, the compress statement will always delay for one clock cycle.) For example, suppose we compressed the blackjack fragment shown above. First, newcard would be assigned the value of the binary decoding of the input in no time. Then, since no time has passed, execution will continue and newcard will be assigned the value 10 if the binary decoding was between 11 and 13. At this point, the body of the compress has terminated and its effect is to assign the value of the dealt card to newcard in zero time. So the compress statement will delay for one clock cycle and then assign this value to newcard.
The compress statement does not effect the loop timing rules. In particular, time can still pass within the compress if the statement contains a loop whose body executes in no time. Since variable assignment takes no time, it is very likely that the body will execute in no time.

There are a few restrictions that are placed upon the statement that is to be compressed. The statement cannot contain any parallel, skip, or delay statements. Moreover, exit and break cannot be used to jump out of the compress.

3. Sample Programs

In order to illustrate the use of SML as a design tool, we have included two examples from well known textbooks on digital design. The first of these programs implements a soda machine controller similar to the one described in [6]. The controller accepts nickels, dimes, quarters and half-dollars until 30 cents have been deposited, whereupon the machine dispenses a can of soda and gives the correct change. There is also a coin release button that will force the controller to return all of the change already entered.

The type of coin received is indicated by two inputs, C1 and C0. The meaning of these two lines is as follows:

<table>
<thead>
<tr>
<th>C1</th>
<th>C0</th>
<th>Type of Coin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nickel</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>dime</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>quarter</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>half-dollar</td>
</tr>
</tbody>
</table>

When the control signal COIN-PRESENT is high for one clock cycle, a coin has been dropped into the collection box. (The type of the coin is specified as above.) The only other input is COIN-EJECT, which goes high when the user presses the coin release button. The sensors are designed so that COIN-PRESENT is disabled whenever COIN-EJECT is high.

The soda machine controller has three outputs: READY, DROP-SODA, and EJECT-NICKEL. When READY is high, the machine is ready to accept another coin. COIN-PRESENT will never go high while READY is low. DROP-SODA should go high for one clock cycle whenever a can of soda is sold. EJECT-NICKEL should go high for one clock cycle whenever a nickel is to be given to the user. The SML program that implements this controller is given in figure 3-1.

A few comments are necessary to explain the operation of this program.

*Lines 3-5:* In addition to declaring the inputs and outputs, an internal integer sum is declared. This integer will store the total amount of money received so far.

*Lines 7-9:* Wait is a macro that delays until its parameter becomes true.
program changer;
input COIN-PRESENT, C1, C0, COIN-EJECT;
output DROP-SODA, EJECT-NICKEL, READY = true;
integer sum(8):

procedure wait(aap)
while (loop) do loop skip endloop
endpro

procedure decode()
switch
case (C1 & C0): sum := sum + 6; break;
case (C1 & C0): sum := sum + 10; break;
case (C1 & C0): sum := sum + 25; break;
case (C1 & C0): sum := sum + 50; break;
case default: sum := sum + 0; break;
endswitch
endpro

loop
parallel
loop if COIN-EJECT then break endif endloop
||
loop
wait(COIN-PRESENT):
compress
decode():
if (sum > 30) then lower (READY) endif
endcompress:
if (sum > 30) then
parallel raise (DROP-SODA) || sum := sum + 30 endparallel:
lower (DROP-SODA):
while sum > 0 do loop
parallel
raise (EJECT-NICKEL) || sum := sum + 5 endparallel:
endloop:
parallel lower (EJECT-NICKEL) || raise (READY) endparallel
endif
endloop:
endparallel:
parallel
lower (READY) || lower (DROP-SODA) || lower (EJECT-NICKEL)
endparallel:
while sum > 0 do loop
parallel
raise (EJECT-NICKEL) || sum := sum + 5 endparallel:
endloop:
parallel lower (EJECT-NICKEL) || raise (READY) endparallel
endloop
endpro

Figure 3-1: The Soda Machine Controller Program

Lines 11-18: Decode is a macro that examines C1 and C0 and updates sum accordingly.

Line 22: This loop monitors the COIN-EJECT signal. If it should go high, we immediately break out of the parallel and go to line 42.

Lines 24-40: This loop is the heart of the program. Once a coin has been received (COIN-PRESENT is high at line 25), we add it to the sum (line 27) and delay other coins if the machine is about to sell a soda (line 28). If thirty cents have been received, we sell a soda (lines 31 and 32) and give the correct change (lines 33-37). Now we can accept another coin (line 38).

Lines 42-50: If COIN-EJECT goes high, we break out of the parallel and end up at this routine. Since the machine might have been selling soda or returning change, we reset DROP-SODA and EJECT-NICKEL and disable the reception of new coins (line 43). Then, as long as there is money to be returned, we eject nickels (lines 45-49). Now we are ready to accept another coin, so we raise READY and loop to the start of the parallel.
This program compiled into a 80 state machine in approximately 0.8 seconds of CPU time on a VAX 11/780.

Our second example is a simple blackjack player similar to the one in [11]. This controller is told which cards are dealt to it and it displays the running total. If the total is less than 21, it also indicates whether it would like another card or not. (The program uses the standard rules for the dealer: it always takes a card on 16 or less and it always refuses a card on 17 or more.)

The type of card is indicated by four inputs, C3, C2, C1, and C0, that are decoded as a binary number. An ace is 1, a jack is 11, a queen is 12, and a king is 13. The program treats an illegal input (0 or more than 14) as a face card. CARD-READY is a control input that is high when the card inputs are valid. There is another input, RESET, that is set high to restart the game after a hand has been finished.

The machine's score is indicated by five outputs, S4, S3, S2, S1, and S0, that are decoded as a binary number. (In blackjack, face cards count as 10, number cards count as their value, and aces count as either 1 or 11.) There are three status outputs: HIT is high if the machine wants another card, STAND is high if the machine has 21 or less and does not want another card, and BUST is high if the machine has 22 or more. The program that implements this controller is given in figure 3-2.

A few comments are necessary to explain the operation of this program.

**Lines 3-6:** In addition to declaring the inputs and outputs, three other variables are also declared. ACE11 is an internal boolean that is high if the present score includes an ace that is being counted as 11. Score is an integer that is the total value of the cards received, and card is a temporary integer that is the value of the current card being received.

**Lines 8-10:** Wait is a procedure that does nothing until its argument becomes true.

**Lines 12-22:** Get card is the procedure that adds the current card to the total. Line 13 decodes the input as a binary number, and line 14 sets the cards value to 10 if it is a face card (or an error). Then, if the card is an ace and we can count it as 11 without busting, we add 11 to the total and indicate that we are counting an ace as 11 (lines 16-17). Otherwise, we just add the card's value to the total (line 19). Since the card's value is no longer needed, we set it to zero (line 19).

**Lines 24-30:** Display simply outputs the current score.

---

2 This isn't actually necessary, since the SML compiler contains a finite state machine minimizer that would determine that the value of card doesn't matter after it has been added to score. However, compilation is much faster if this variable is cleared. (Unfortunately, SML variables are global, and what is really needed is an integer that is local to get card.)
program blackjack;

input RESET, CARD-READY, C3, C2, C1, CO;
output HIT = true, STAND, MUST, S4, S3, S2, S1, SO;
internal ACE11;
integer score[8], card[4];

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

procedure getcard()
  card := int(C3) + 2 * int(C1) + 4 * int(C2) + 8 * int(CS);
  if (card == 0) || (card > 10) then card := 10 endif;
  if (card == 1) && (score < 11) then
    score := score + 11;
  else
    score := score + card
  endif;
  card := 0
endproc

procedure display()
  S0 := score[0];
  S1 := score[1];
  S2 := score[2];
  S3 := score[3];
  S4 := score[4]
endproc

loop
  input (CARD-READY):
  compress:
  lower(HIT);
  getcard();
  display();
  switch
  case score > 21 && ACE11:
    lower(ACE11);
    score := score - 11;
  display();
  case score > 21:
    raise(MUST);
    break;
  case score > 16:
    raise(STAND);
    break;
  default:
    raise(HIT);
  endswitch;
  endcompress:
  if (MUST || STAND) then
    endi:
    endloop:
    wait (RESET):
    parallel
    raise(HIT) || lower(STAND) || lower(MUST)
    ||
    score := 0 || lower(ACE11)
    ||
    lower(S0) || lower(S1) || lower(S2) || lower(S3) || lower(S4)
  endparallel
endloop
endloop
endproc

Figure 3-2: The Blackjack Player Program

Lines 32-66: This is the main program loop. The machine waits with HIT high until a card is received (line 34). We no longer need a card, so HIT is lowered (line 36). The value of the card is added to the total and the new total is displayed (lines 37-38). (Notice that all of these actions are done simultaneously because of the compress on line 35.) The switch statement (lines 39-52) then determines the next action.

Lines 40-43: If the total is over 21 but we have counted an ace as 11 instead of 1, the machine will recount the ace as 1 (line 41) and subtract 10 from the score (line 42). The new score is then displayed (line 43) and we fall through to test the next condition of the switch.
Lines 44-46: If the total is over 21, the machine is busted, so BUST is raised.

Lines 47-49: If the total is over 16, the machine follows the dealer's rules and refuses another card by raising STAND.

Lines 50-51: If the total is 16 or less, the machine wants another card, so HIT is raised.

Lines 54-55: If the machine is busted or doesn’t want any more cards, the game is over, so we exit the loop. Otherwise, the loop restarts and we wait for another card.

Lines 58-65: Once the game is over, we wait for RESET to start a new game (line 58). Once reset is received, all of the internal and output variables are reset to their original values, and we start a new game.

This program compiled into a 32 state machine in approximately 2.3 seconds of CPU time on a VAX 11/780.

4. Interface with Temporal Logic Verifier

In this section we give a more complicated example to illustrate how our language can be used in conjunction with the temporal logic model checker described in ([2], [5]) in order to obtain a program that is guaranteed to satisfy certain timing properties. We refer the interested reader to these papers for a full description of the version of temporal logic that we use for specification and for a discussion of how the verifier works. Furthermore, since the example in this section is considerably longer than the examples in the previous section, we will not attempt to give a line by line description of how it works. The example that we consider is a simple bus protocol in which a master process sends data to some number of slave processes for special computation or perhaps some type of I/O operation. A system diagram is shown in Figure 4-1. The program for the case with three slave processes is shown in Figure 4-2. The bus consists of several control lines (REQ[4], GO[4], RCVR[2], ACK, and DV) and some unspecified number of data lines. Since the data lines do not effect the flow of control in this example, they are not explicitly mentioned in the text of the program. The bodies of the processes are specified as SML procedures and will be described separately below.

The bus controller arbitrates access to the bus and guarantees that only one of the processes is using it at any given time. Associated with each process is a REQ line and a GO line. When a process wants to use the bus it raises its REQ line and waits for the corresponding GO line to become high. The bus controller scans the request lines until it finds one that it high, it then raises the appropriate GO line and waits until the REQ line goes low indicating that the current communication has been completed. It then lowers the GO signal and continues its scan for another REQ line that is high.
The master process first checks if there is a job pending and if one of the slave processes is not busy. If so, it will send the data for the job to the idle slave process. It then checks to see if any slave process has finished a job and is trying to send a completion message. The loop is then repeated. We assume that a job is pending whenever job req is different from job started. Since new job is a program input, the if statement at the beginning of the master procedure will cause this condition to change nondeterministically. The local bit array busy[i] is used to indicate whether the i\textsuperscript{th} slave is currently processing a job or not.

In procedure send(i), the master first checks to see if the i\textsuperscript{th} slave is busy and if there is a pending job. If so, it requests permission to use the bus and waits for a GO signal from the bus controller. When it receives permission to use the bus, it sets RCVR to indicate the identity of the receiving process (the i\textsuperscript{th} slave). It also raises the DV signal to indicate that the data on the bus is valid. When it receives acknowledgment that the slave has read the data, it lowers the REQ and DV lines and waits for the bus controller to lower the GO signal and for the slave process to lower its ACK signal. Finally, it sets busy[i] to indicate that the i\textsuperscript{th} slave is busy and inverts job started to indicate that the pending job has been started.

Procedure receive is even simpler; the master checks to see if any slave process is trying to send an indication that it has completed a job--this will be the case whenever the condition ((RCVR = M) & DV) holds. The master then determines the identity of the slave process and lowers the appropriate entry in the busy array. It acknowledges receipt of the message by raising the ACK signal and waits until the slave process lowers its REQ line.
program master;slave:

--components of bus
integer REQ[4], GO[4], NCVR[2];
integer ACE, GW;

--variables local to master
integer jobreq, jobstarted;
input newjob;
integer busy[4];

--variables local to slave processes
integer running[4], running idle, running finished;
input finished, finished idle, finished finished;

procedure wait(sap) ... endproc
procedure buscontroller ... endproc
procedure send[i] ... endproc
procedure receive ... endproc
procedure slave[1, running, finished] ... endproc
procedure master ... endproc

parallel

writer

|| master

|| idle

|| slave[SL, running idle, finished idle]
|| slave[SL, running idle, finished finished]
|| slave[SL, running idle, finished finished]

endparallel

endproc

procedure buscontroller

loop

if REQ[i] then
  raise(GO[i]);
  wait(RCVR[i]);
  lower(GO[i]);
endif;

if REQ[1] then
  raise(GO[1]);
  wait(1RCVR[1]);
  lower(GO[1]);
endif;

if REQ[2] then
  raise(GO[2]);
  wait(2RCVR[2]);
  lower(GO[2]);
endif;

if REQ[3] then
  raise(GO[3]);
  wait(3RCVR[3]);
  lower(GO[3]);
endif;

if REQ[4] then
  raise(GO[4]);
  wait(4RCVR[4]);
  lower(GO[4]);
endif;

endloop

endproc

Figure 4-2: Simple Bus Protocol

Figure 4-3: The bus controller.

Each slave process consists of two subprocesses. One process waits for a message from the master indicating that there is a new job for the slave to process. When it receives such a message it raises its local copy of running to indicate that the job is being processed and sends an acknowledgment to the master. Once running becomes high, the other subprocess will wait until the local processing of
the job is finished. When this happens, it notifies the master that the job has been completed. It follows essentially the same protocol described previously to gain control of the bus inorder to send the completion message.

The bus controller, the master process, and the slave processes would each be compiled separately for an actual implementation in hardware. However, for analysis with the temporal logic model checker all of the processes must be compiled together. This results in a Moore machine with 1372 states. In many applications it is important to determine whether the state machine generated by the compiler correctly implements certain timing properties. In our example an obvious question to ask is whether the master will always be able to successfully start a pending job on one of the slaves. Of course, we can't guarantee that this property is satisfied unless we assume that when a slave starts executing a job, the job will eventually terminate. We show below that both the property that we want
procedure slave(1, running, finished)
  parallel loop
    if (NCVR = 1) & DV then
      parallel
        raise(running) || raise(ACE)
      endparallel;
      wait(REQ[1]);
      lower(ACE);
    endif
  endloop
  if running then
    parallel
      wait(finished);
      lower(running) || raise(REQ[1])
    endparallel;
    wait(ACE);
    parallel
      NCVR := N || raise(DV)
    endparallel;
    wait(ACE);
    parallel
      lower(REQ[1]) || lower(DV)
    endparallel;
    wait(REQ[1] & LACE);
  endif
endparallel
endproc

Figure 4-7: Code for Slave

to check and the assumption on job termination can be easily expressed in temporal logic and verified
using the model checking program.

The logic that we use for specification is a propositional, branching-time temporal logic called CTL
or Computation Tree Logic [5]. Typical operators include G f, which holds in the present state
provided that f holds globally in all possible computations paths starting from the present state, and F
f, which holds in the present state provided that f inevitably holds in the future in all possible
computations. Our model checking program [2] determines whether a temporal logic formula is true
or not by traversing the state graph of the state 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.

The property that we want to check can be expressed by the CTL formula:

\[ G((jobreq \neq jobstarted) \rightarrow (jobstarted \rightarrow F \neg jobstarted)) \land (\neg jobstarted \rightarrow F jobstarted) \].

The formula \( f \neq g \) is true provided the truth value of \( f \) is different from the truth value of \( g \). Thus,
formula 1 can be paraphrased in English as follows: it is invariantly true that if \( jobreq \) is different from
\( jobstarted \) (indicating that there is a pending job), then the truth value of \( jobstarted \) will always be
eventually inverted (indicating that the job has been successfully started on a slave). In order to
express the termination assumption for slave computations, we must run the verifier with three
fairness constraints, one for each slave process. Any CTL formula can be used as a fairness
constraint. A computation path is fair if each fairness constraint holds infinitely often on the path.
The verifier will only check \emph{fair} computation paths in determining whether a CTL formula is true or not. In this case a path is fair if for each slave process, \(\neg\) running or finished holds infinitely often.

Figure 4-8 shows the output produced by our verifier with the CTL formula and fairness constraints discussed in the preceding paragraph. The verifier determines that the formula does not hold and gives a counterexample. Time is measured in \(1/60\) of a second, so our program is able to find a bug in less than 18 seconds! By examining the counterexample we can quickly determine the source of the problem. The master starts a job on one of the slaves. The slave finishes the job and gets control of the bus to send its completion message to the master. In the meantime a new job arrives and the master tries to start the job on another slave. The master also requests to use the bus and waits for the GO signal giving it permission. Since the slave has control of the bus and is waiting for the master to acknowledge receipt of its completion message, this causes a deadlock.

The problem can be fixed by slightly modifying the send procedure as shown in Figure 4-9. Instead of waiting for the GO, the master relinquishes its attempt to gain control of the bus if its request is not immediately granted. If the program is recompiled with the new version of send, a Moore machine with 1474 states is obtained. When the new state machine is checked using the same specification, the verifier determines that the CTL formula is satisfied in approximately 13 seconds. By checking other desirable properties in a similar manner, it is possible to obtain a high degree of confidence in the correctness of the program.

5. Compilation of SML Programs

In section 2, we informally described how each of the SML statements changes the current assignment of values to variables. However, SML also has a formal semantics, parts of which are described here. An SML \emph{program state} is an assignment of values to variables and an SML statement that is to be executed. The formal semantics of SML is an operational semantics that describes how one program state can be transformed into another program state, depending on the state of the input signals. The semantics consists of a set of \emph{conditional rewrite rules} that describe the possible transformations as well as the amount of time that each transformation takes. The rewrite rules are defined so that there is always a unique transformation to a new state that takes one time unit.

In order to illustrate the use of rewrite rules, here are four typical rules along with an explanation of each.

\[
\begin{align*}
S[E]s &= true \\
&\frac{}{(if \; E \; then \; C \; endif, s) \xrightarrow{\delta} (C, s)}
\end{align*}
\]

\(S\) is the meaning function for expression. So this rule states that if expression \(E\) is true when the variable assignment is \(s\), then an if statement can be transformed into the body of the if in zero time.

\[
\begin{align*}
\langle\text{raise} (l), s\rangle &\xrightarrow{1} \langle s, s[l\rightarrow\text{true}]\rangle
\end{align*}
\]
CTL MODEL CHECKER (version B1.0)

Reading masterslave...
Fairness constraint: ¬running1 | finished1.
Fairness constraint: ¬running2 | finished2.
Fairness constraint: ¬running3 | finished3.
Fairness constraint:

time: 4349
| = AG ((jobreq req jobstarted) → (jobstarted → F ¬jobstarted) & (¬jobstarted → F jobstarted)).
The formula is FALSE.
State 0-0:
State 0-1: newjob
State 1-0: jobreq
State 2-0: jobreq REQ0
State 3-0: jobreq REQ0 GO0
State 4-0: DV jobreq REQ0 GO0 RCVR0
State 5-0: ACK DV jobreq running1 REQ0 GO0 RCVR0
State 6-0: ACK jobreq running1 GO0 RCVR0
State 8-0: jobreq running1 RCVR0
State 10-2: finished1 jobreq running1 RCVR0 busy1
State 14-0: jobreq jobstarted REQ1 RCVR0 busy1
State 19-0: jobstarted REQ1 GO1 RCVR0 busy1
State 25-0: DV jobstarted REQ1 GO0 busy1
State 25-0: DV jobstarted REQ1 REQ0 GO1 busy1
...

time: 1072

Figure 4-8: Error in Bus Protocol

procedure send(1)
    if (loop[1] & (jobreq != jobstarted)) then
        raise(REQ[1]);
        wait(GO[1] | (RCVR == H) & DV);
        if (GO[1]) then
            lower(REQ[1]);
        else
            parallel
            proc := H || RCVR := 1 || raise(DV)
            parallel
            wait(ACK);
            parallel
            lower(REQ[1]) || lower(DV)
            parallel
            wait(1GO[1] & (ACK);
            raise(busy[1]));
            lower(jobstarted);
        endif
    endif

Figure 4-9: New Procedure Send

This rule indicates that a raise statement can be transformed into the empty statement in one time
unit. In addition, the variable assignment is changed so that the variable that is raised is now true.

$$<C,s> \not\rightarrow <s,s>$$

$$<\text{loop } C \text{ endloop,s}> \not\rightarrow <\text{loop } C \text{ endloop,s}>$$

If statement $C$ does nothing in zero time, then a loop statement with $C$ as the body does nothing in
one time unit.
\[
\langle C_{1};a \rangle \triangleright \langle e, s \rangle \quad \langle C_{2};a \rangle \Downarrow \langle e, s' \rangle
\]

\[
\langle \text{parallel} \ C_{1} \parallel \ C_{2} \ \text{endparallel},s \rangle \Downarrow \langle e, s' \rangle
\]

If \( C_{1} \) does nothing in zero time and \( C_{2} \) terminates after doing something in one time unit, then executing these statements in parallel has the same effect as executing \( C_{2} \).

The operational semantics for SML naturally leads to a simple compilation algorithm. The initial program state is the body of the program and the variable assignment described by the variable declarations. Then, for each possible state of the inputs, the rewrite rules can be applied to find the unique next program state. This procedure can be repeated for each new program state until the entire state machine is created.\(^3\)

The real SML compiler uses an algorithm very similar to this one. However, by analyzing the structure of the program and choosing an appropriate representation for the SML statement to be executed, the application of rewrite rules can be made very efficient. By examining the placement of parallel statements in the program, the maximum number of concurrent activities can be found. Then, the SML statement to be executed will be represented by an array of pointers into a global parse tree, one for each possible concurrent activity. Each statement (or sub-statement) in the program is statically assigned an index into this array, so that a pointer to this statement will only occur in that array index. Now, the context of the program limits the number of applicable rewrite rules for each statement, we can speed the application of rules by writing a pseudo-code program that applies the correct rewrite rule. (For example, if the condition of an if statement is false, there is a rewrite rule that transforms the if to the empty string in zero time. But since there is also a rule that transforms two statements in sequence into the second statement if the first takes no time, the pseudo-code program will combine these transformations into one.) Therefore, the next state can be found using the following procedure:

```
procedure nextstate (places, values)
begin
    integer i;
    for i := 1 to MaxPlaces do
        if places[i] = nil then
            execute-p-code (i, places, values);
        return <places, values>
    end
end
```

We should note that the compress statement is very easy to implement in this scheme. There is a special pseudo-instruction, END, that is used to stop the application of rewrite rules after a rule indicates a delay. Normally, an END instruction is placed after every variable assignment so that time will pass. However, if the variable assignment is within a compress statement, we simply omit the END instruction.

\(^3\)Since all variables are finite state, there are only a finite number of variable assignments. Moreover, all programs are finite length, so there are only a finite number of possible program states, so this algorithm must terminate.
In order to increase efficiency, the expression evaluator keeps track of which inputs are examined while finding a successor. If there are some inputs that are not tested, then input states that differ from the current input only in these inputs will have the same successor, so these inputs do not need to be considered.

The output of the SML compiler is a finite state machine in FIF (FSM Intermediate Format). An example of this format is given in figure 5-1. FIF is accepted as input by a program named aflc (A FSM Compiler) which will produce a ROM, PLA, or PLA based implementation of the state machine. aflc can produce output that is compatible with the Berkeley tools kiss and presto as well as several other formats.

NAME = blackjack;
STATES = 32;
CUBES = 308;
#0 10000000
X11001 10
X11000 9
X10111 8
X10110 7
X10101 6
X10100 5
X10011 4
X10010 3
X10001 2
X10000 1
X11011 1
X11010 1
X11000 0
#1 10001010
X11001 20

Figure 5-1: Sample Output of the SML Compiler

This compilation procedure is linear in the number of states in the machine and exponential in the number of input variables. But as the program in figure 5-2 illustrates, the number of states in the machine can be exponential in the number of output variables, the procedure must be exponential in the number of output variables as well. Furthermore, the application of rewrite rules in order to find a successor state can be linear in the size of the program. (For example, if the program consists of a large loop, it might be necessary to apply the rewrite rules to every statement in the loop before discovering that the loop body takes no time.)
6. Other Finite State Machine Languages

The first research on using high level languages for describing state machines was probably due to Parnas [8] in 1966. Unfortunately, his work on this subject appears to have been largely forgotten. Perhaps this was because good tools for converting state machines into circuits were not widely available in 1966. We did not learn about his work until an early version of this paper had already been completed. His language, SFD-Algol, was quite powerful—it even contained a statement somewhat like our compress statement. His language lacks modern control structures like the parallel execution statement, which we find quite useful. Furthermore, no tools were available for verifying the correctness of state machines produced by his compiler.

The development of SML was heavily influenced by the language ESTEREL [1]. In ESTEREL, neither computation, control flow nor variable assignment takes any time. Time passes only when the program is explicitly waiting for an external event. This timing model can cause “temporal paradoxes”, such as the one in the following ESTEREL statement:

```
do emit s upto s
```

The ESTEREL do .. upto construction executes the body (in this case, "emit s") until the signal s is present. The emit statement asserts the specified signal (in zero time). Therefore, s is emitted if not present and is not emitted if present. But since control flow and emission takes no time, this statement is nonsense. Although the ESTEREL compiler detects this problem and indicates an error, we believe that the possibility of problems like this one make it difficult to write ESTEREL programs. In order to avoid these difficulties, we require variable assignments to take one time unit. Although this makes SML less flexible than ESTEREL, we believe that it is much easier to write correct programs in SML than in ESTEREL.4

Many other languages, such as AMAZE, CUPL, and SLIM [10], have been designed to describe

---

4In order to introduce more flexibility, we added the compress statement to SML. However, the restrictions that are placed on this statement (no parallel, exit, or break can be compressed) insure that no temporal paradoxes can occur in SML.
finite state machines. However, these languages deal with finite state machines on a very low level. All of these languages require an explicit description of the state-transition behavior of the machine which is to be implemented. On the other hand, SML allows the behavior of the machine to be described algorithmically, with the compilation process extracting the state-transition behavior. These languages can be thought of as state machine assembly languages, but SML is a true high level language. (As a matter of fact, it would be a simple matter to modify the SML compiler to produce its output in any of these languages.)

7. Conclusion

In this paper, we have described a high level language for specifying machines (SML) and illustrated its use with several examples. Although the compilation procedure for the language is exponential in worst case, the compiler is fast enough so that SML is still quite useful for the design of small (≤ 1000 state) finite state machines. The output of the SML compiler can be used by the Berkeley VLSI design tools to layout the finite state machines as ROMs, PLAs, or PALs. Furthermore, we have interfaced the SML compiler with a temporal logic theorem prover (\cite{2,5}) that can assist in the debugging of SML programs. We believe our system is the first to provide this type of verification facility. Several groups within the Computer Science Department at CMU are actively experimenting with SML for rapid design of hardware controllers.

There are a number of obvious extensions that would make SML even more useful for designing and verifying circuits. For example, in specifying the bus controller in Section 4 essentially the same segment of code had to be duplicated several times. It should be quite easy to provide a syntactic mechanism within the language to eliminate such tedious repetition. Local variables associated particular processes and a more sophisticated type structure including enumeration types would also make SML programs easier to read and to understand. A more difficult issue to deal with is nondeterminism. Currently, SML processes are completely synchronous. In practice, however, it is important to reason about processes that run on different clocks or execute asynchronously \cite{7}. More research is needed to handle this problem within our current framework. Finally, it seems reasonable that the bus protocol should work correctly with an arbitrary number of slave processes provided that the obvious changes are made to the program. In general, it may be possible to reason about a system of \( N \) identical processes by reducing the number of processes to some small fixed value and using the temporal logic model checker. Some preliminary research in this direction is described in \cite{9}.

References


