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.

In order to assist with the design of finite state machines, we have designed a programming language named SML (State Machine Language). In addition to being useful for design, SML can also be a documentation aid, since it provides a succinct notation for describing complicated finite state machines. A program written in SML can be compiled into a state transition table that can then be implemented in hardware using an appropriate design tool. The resulting state transition table can also be given to a *temporal logic model checker* ([2] [5]) that allows certain properties of the state machine to be automatically verified. This is discussed extensively in some of our other papers ([3] [4]), and will not be mentioned here.

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 my 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 to 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:

Table of Contents

List of Figures

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*[1]

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

[1] SML procedures are actually implemented as macros.

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;
integer X[3] = 5, Y[2] = 7, Z[5];

As a result of these declarations:

- 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 though" 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 make 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:
    input S3, S2, S1, S0;
    integer newcard[4] = 0;
        .
        .
        .
newcard := int(S0) + 2 * int(S1) + 4 * int(S2) + 8 * int(S3);
if (newcard > 10 & newcard < 14) 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 written several sample programs. 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:

C1	C0	Type of Coin
0	0	nickel
0	1	dime
1	0	quarter
1	1	half-dollar

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.

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 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

```
1       program changer;
2
3       input COIN-PRESENT, C1, C0, COIN-EJECT;
4       output DROP-SODA, EJECT-NICKEL, READY = true;
5       integer sum[8];
6
7       procedure wait(exp)
8           while !(exp) do loop skip endloop
9       endproc
10
11      procedure decode()
12          switch
13              case !C1 & !C0: sum := sum + 5; break;
14              case !C1 & C0: sum := sum + 10; break;
15              case C1 & !C0: sum := sum + 25; break;
16              default: sum := sum + 50; break;
17          endswitch
18      endproc
19
20          loop
21              parallel
22                  loop if COIN-EJECT then break endif endloop
23              ||
24                  loop
25                      wait(COIN-PRESENT);
26                      compress
27                          decode();
28                          if (sum > 30) then lower (READY) endif
29                      endcompress;
30                      if (sum > 30) then
31                          parallel raise (DROP-SODA) || sum := sum - 30 endparallel;
32                          lower (DROP-SODA);
33                          while sum > 0 do loop
34                              parallel
35                                  raise (EJECT-NICKEL) || sum := sum - 5
36                              endparallel
37                          endloop;
38                          parallel lower (EJECT-NICKEL) || raise (READY) endparallel
39                      endif
40                  endloop
41              endparallel;
42              parallel
43                  lower (READY) || lower (DROP-SODA) || lower (EJECT-NICKEL)
44              endparallel;
45              while sum > 0 do loop
46                  parallel
47                      raise (EJECT-NICKEL) || sum := sum - 5
48                  endparallel
49              endloop;
50              parallel lower (EJECT-NICKEL) || raise (READY) endparallel
51          endloop
52      endprog
```

Figure 3-1: The Soda Machine Controller Program

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 [8]. 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: **Getcard** 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).[2]

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

Lines 32-66: This is the main program loop. The machine waits with HIT high until a card is received (line

[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 **getcard**.)

```
1       program blackjack;
2
3       input RESET, CARD-READY, C3, C2, C1, C0;
4       output HIT : true, STAND, BUST, S4, S3, S2, S1, S0;
5       internal ACE11;
6       integer score[5], card[4];
7
8       procedure wait(exp)
9           while !(exp) do loop skip endloop
10      endproc
11
12      procedure getcard()
13          card := int(C0) + 2 * int(C1) + 4 * int(C2) + 8 * int(C3);
14          if (card == 0) | (card > 10) then card := 10 endif;
15          if (card == 1) & (score < 11) then
16              score := score + 11;
17              raise(ACE11)
18          else
19              score := score + card
20          endif;
21          card := 0
22      endproc
23
24      procedure display()
25          S0 := score[0];
26          S1 := score[1];
27          S2 := score[2];
28          S3 := score[3];
29          S4 := score[4]
30      endproc
31
32          loop
33              loop
34                  wait (CARD-READY);
35                  compress
36                      lower(HIT);
37                      getcard();
38                      display();
39                      switch
40                          case score > 21 & ACE11:
41                              lower(ACE11);
42                              score := score - 10;
43                              display();
44                          case score > 21:
45                              raise(BUST);
46                              break;
47                          case score > 16:
48                              raise(STAND);
49                              break;
50                          default:
51                              raise(HIT);
52                      endswitch;
53                  endcompress;
54                  if (BUST | STAND) then
55                      exit;
56                  endif;
57              endloop;
58              wait (RESET);
59              parallel
60                  raise(HIT) || lower(STAND) || lower (BUST)
61              ||
62                  score := 0 || lower (ACE11)
63              ||
64                  lower (S0) || lower (S1) || lower (S2) || lower (S3) || lower (S4)
65              endparallel
66          endloop
67      endprog
```

Figure 3-2: The Blackjack Player Program

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. 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 *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 *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.

$$\frac{\mathcal{E}[\![E]\!]s = true}{\langle \textbf{if E then C endif},s\rangle \xrightarrow{0} \langle C,s\rangle}$$

$\mathcal{E}$ 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.

$$\langle \textbf{raise (I)},s\rangle \xrightarrow{1} \langle \epsilon, s[I \mapsto true]\rangle$$

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.

$$\frac{\langle C,s\rangle \xrightarrow{0} \langle \epsilon,s\rangle}{\langle \textbf{loop } C \textbf{ endloop},s\rangle \xrightarrow{1} \langle \textbf{loop } C \textbf{ endloop},s\rangle}$$

If statement C does nothing in zero time, then a **loop** statement with C as the body does nothing in one time unit.

$$\frac{\langle C_1,s\rangle \xrightarrow{0} \langle \epsilon,s\rangle \quad \langle C_2,s\rangle \xrightarrow{1} \langle \epsilon,s'\rangle}{\langle \textbf{parallel } C_1 \parallel C_2 \textbf{ endparallel},s\rangle \xrightarrow{1} \langle \epsilon,s'\rangle}$$

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
```

[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.

```
        if places[i] ≠ nil then
            execute-p-code(i, places, values);
    return <places, values>
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!

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 4-1. FIF is accepted as input by a program named *afc* (A FSM Compiler) which will produce a ROM, PLA, or PLA based implementation of the state machine. *Afc* can produce output that is compatible with the Berkeley tools *KISS* and *presto* as well as several other formats.

```
NAME = blackjack;
STATES = 32;
CUBES = 306;
INPUTS = RESET.H, CARDREADY.H, C3.H, C2.H, C1.H, C0.H;
MOORE-OUTPUTS = HIT.H, STAND.H, BUST.H, S4.H, S3.H, S2.H, S1.H, S0.H;
#0      10000000
X11001  10
X11000  9
X10111  8
X10110  7
X10101  6
X10100  5
X10011  4
X10010  3
X10001  2
X10000  1
X1110X  1
X11X1X  1
X0XXXX  0
#1      10001010
X11001  20
...
```

Figure 4-1: 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 4-2 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.)

```
program exp;

output S0,S1,...,Sn-1,Sn;

    loop
        invert (S0);
        if !S0 then
            invert (S1);
            if !S1 then
                ...
                    if !Sn-2 then
                        invert (Sn-1);
                        if !Sn-1 then invert (Sn) endif
                    endif
                ...
            endif
        endif
    endloop
endprog
```

Figure 4-2: A Program With An Exponential Number of States

5. Other Finite State Machine Languages

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 [7], have been designed to describe 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

[4] In 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.

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.)

6. Conclusion

In this paper, we have described a finite state language named SML and illustrated its use with two examples. Although the compilation procedure is exponential, the compiler is fast enough that we believe that SML can still be a useful tool for the design of small (< 1000 state) finite state machines. Furthermore, we have interfaced our SML compiler with a temporal logic theorem prover [2] that can assist in the debugging and verification of SML programs. The output of the SML compiler can also be used by the Berkeley VLSI design tools to layout the finite state machines as either a ROM, PLA, or PAL.

References

[1] G. Berry and L. Cosserat.
The ESTEREL Synchronous Programming Language and its Mathematical Semantics.
Technical Report, Ecole Nationale Superieure des Mines de Paris, 1984.

[2] M.C. Browne.
An Improved Algorithm for the Automatic Verification of Finite State Machines Using Temporal Logic.
1985.
Unpublished Technical Report.

[3] M. Browne, E. Clarke, D. Dill.
Automatic Circuit Verification Using Temporal Logic: Two New Examples.
In *IEEE International Conference on Computer Design: VLSI and Computers.* Port Chester, NY, October, 1985.

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

[5] E.M. Clarke, E.A. Emerson, A.P. Sistla.
Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications: A Practical Approach.
In *Tenth ACM Symposium on Principles of Programming Languages.* Austin, Texas, 1983.

[6] W.I. Fletcher.
An Engineering Approach To Digital Design.
Prentice-Hall, Inc., 1980.

[7] J.D. Ullman.
Computational Aspects of VLSI.
Computer Science Press, 1984.

[8] D. Winkel and F. Prosser.
The Art of Digital Design.
Prentice-Hall, Inc., 1980.