Reference Manual for CSML Verification System

CSML Language Reference

M. C. Browne
Carnegie-Mellon University

E. M. Clarke
Carnegie-Mellon University

K. L. McMillan
Carnegie-Mellon University

May 10, 1989

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* [Bro85,EC83] that allows certain properties of the state machine to be automatically verified.

This report describes an extended version of SML called CSML, for Compositional State Machine Language. CSML is a strict extension of SML; all SML programs should be compiled identically by the CSML compiler. The CSML extensions, however, allow the specification of separate communicating modules. This means that a state machine that has too many states to be implemented directly may be implemented as a number of much smaller modules. It also allows compositional model checking techniques to be used, to reduce the complexity of verifying state machines using a temporal logic model checker [CLM89a,CLM89b].

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 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*[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 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.

[1] SML procedures are actually implemented as macros.

- 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.)

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.

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**.)

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.

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.

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.

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!)

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.

```
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 1: Card Decoding in a Blackjack Program

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 fragment from a blackjack dealing program in figure 2.3.

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 The CSML extensions

In this section we describe two extensions we have made to SML which allow the hierarchical definition and interconnection of modules. A *process* in CSML has the following syntax:

process *identifier;*
  declaration list
  statement list
endproc

A process compiles into a separate Moore machine which communicates with other processes running concurrently (in lockstep). The *statement list in a program or process may be replaced by a list of processes. The scoping of variables in CSML is similar to that of PASCAL. A process may access the variables of any process which lexically contains it (all processes may access the variables of the main program). In order to alter a variable of another process, however,*

a process must declare that variable as an output in its declaration list. By requiring that no variable may be declared as an output by more than one process, we insure that CSML processes can be implemented as communicating Moore machines. Variables which are not declared as outputs by any process are considered inputs to the program.

The other way in which CSML extends SML is the processtype statement, which defines a reusable process type. The processtype statement may appear in the declaration list of a program or process, and has the following form:

processtype *identifier (formal parameter list);*
  declaration list
  statement list
endtype

A process type is instantiated as a process by a statement of the following form:

process *process identifier : processtype identifier (actual parameter list);*

The scoping rules for process type identifiers and variables referenced in process types are the same as for variables referenced in processes. In particular, since lexical scoping is used, an instantiated process operates in the context in which it was defined, not in the context in which it is instantiated.

Figure 2 gives a simple example of a CSML program—a system composed of a producer module and a consumer module which synchronize using a four-phase handshake. The formal model of communicating Moore machines which underlies the semantics of CSML is described in [CLM89a].

4 Compilation of SML Programs

The output of the CSML compiler is a set of finite state machines in @i(FIF) (FSM Intermediate Format). An example of this format is given in figure @ref(FIF). 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.

The compiler produces a state machine table in the output file for each process statement in the program. In programs that have a hierarchical module structure, there is a need for a naming convention for referring to nets in the output file. CSML uses the following rules:

- A net corresponding to a variable declared in the main program is referred to by that variable name.
- A net that corresponds to an internal variable of a process is referred to by the variable name, prefixed by the hierarchical name of the process, separated by an underscore character.
- The hierarchical name of a process declared in the main program is the name of the process.
- The hierarchical name of a process nested within another process is the process name prefixed by the hierarchical name of the parent process, separated by an underscore character.

Thus, for example, if variable `c` is declared internal in process `b`, which is declared in process `a`, which is declared in the main program, the net name is `a_b_c`.

In order to allow the temporal logic model checker to be used on CSML programs, there is a separate program called the *state machine composer*. This program takes as input a set of finite state machines generated by the CSML compiler, and produces a single state machine which represents the parallel composition of the modules in the CSML program. For information on running the CSML compiler and state machine composer, see the manual entry for "`csml`".

```
program prodcom;
  output produce,consume;
  internal req,ack;

  processtype Producer(request,acknowledge,produce);
    input request;
    output acknowledge=false,produce=false;
    loop
      while(!request) do loop skip endloop;
      raise(produce); lower(produce);
      raise(acknowledge);
      while(request) do loop skip endloop;
      lower(acknowledge)
    endloop
  endtype

  processtype Consumer(acknowledge,request,consume);
    input acknowledge;
    output request=false,consume=false;
    loop
      raise(request);
      while(!acknowledge) do loop skip endloop;
      raise(consume); lower(consume);
      lower(request);
      while(acknowledge) do loop skip endloop
    endloop
  endtype

  process producer1: Producer(req,ack,produce);
  process consumer1: Consumer(ack,req,consume)
endprog
```

Figure 2: Producer-consumer program