Synthesizable Verilog
syntax and semantics

VFE Project
University of Cambridge Computer Laboratory

Version 0.0

January 28, 1997

The Verilog Formal Equivalence (VFE) Project is funded by the U.K. Engineering and Physical Sciences Research Council.

Contents

Preface

Synthesizable Verilog is a subset of the full Verilog HDL [6] that lies within the domain of current synthesis tools (both RTL and behavioral).

This document specifies a subset of Verilog called SV0. This subset is intended as a vehicle for the rapid prototyping of ideas.

The method chosen for developing a semantics of all of synthesizable Verilog is to start with something too simple – SV0 – and then only to make it more complicated when the simple semantics breaks. This way it is hoped to avoid unnecessary complexity. It is planned to define sequence of bigger and bigger subsets (SV1, SV2 etc.) that will converge to the version of Verilog used in the VFE project[1] at Cambridge.

Different tools interpret Verilog differently: industry standard simulators like Cadence's Verilog XL are based on the scheduling of events. Synthesizers and cycle-simulators are based on a less detailed synchronous next-state semantics.

It is necessary to give an explicit semantics to Verilog to provide a basis for defining what it means to check the equivalence between behavioral prototypes and synthesized logic. In the VFE project, equivalence will be formulated in terms of the cycle based semantics. However, it is hoped eventually to be able to establish that this is consistent with the event semantics used by most simulators. Only a cycle based semantics is given here.

In addition to the immediate goal of defining equivalence between Verilog texts, explicit semantics provide a standard for ensuring that different tools (e.g. simulators and synthesizers) have a consistent interpretation of the language constructs.

[1] VFE stands for Verilog Formal Equivalence. This is our internal name for the EPSRC project entitled *Checking Equivalence Between Synthesised Logic and Non-synthesisable Behavioural Prototypes*.

Some of the features missed out of `SV0` are listed below. Consideration of these omitted features may fatally break the style of semantics given here.

1. The syntax and semantics of expressions is not specified in detail.
2. Module hierarchy is ignored: only a single module is considered.
3. Modules and sequential blocks cannot have local declarations.
4. Vectors, arrays, memories, gates, gate instantiations, drive strengths, delays, and tasks are all omitted.

The semantics is specified by translating the programming constructs to a 'semantic pseudo-code'. The pseudo-code is intended to provide a simpler representation on which to define both the event semantics and the cycle semantics (only the latter is given here, see [2] for an example of the former). It is also hoped to be a first step towards a Verilog/VHDL neutral level (though what, if anything, needs to be added to support VHDL has not been investigated).

The cycle-based semantics given in Chapter 2 derives the state transformations corresponding to cycles by 'symbolically executing' the pseudo-code between timing controls (`@`-constructs). This approach is based on the algorithm underlying David Greaves' *CSYN* compiler [3].

Acknowledgements

This work is funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC) as project GR/K57343. The principal Investigator is Dr. David Greaves.

This method of symbolic execution described in 2.3 is based on the algorithm underlying David Greaves' *CSYN* compiler [3]. The examples here were generated using a program built by Mike Gordon on top of Daryl Stewart's P1364 Verilog parser and pretty-printer [4] which, in turn, is implemented using the syntax processing facilities of Richard Boulton's CLaReT system [1]. Errors in a first draft were pointed out by Daryl Stewart.

We are grateful to Synopsys, Inc. for providing us with their software and for ongoing cooperation in defining the semantics of synthesizable Verilog.

Chapter 1

Syntax

A complete specification in SV0 consists of a single module of the general form:

```
module <module_name> (<port_name>,...,<port_name>);

 function <function_name>;
  input <name>,... , <name>;
  <statement>
 endfunction
   ⋮
 function <function_name>;
  input <name>,... , <name>;
  <statement>
 endfunction

 assign <wire_name> = <expression>
   ⋮
 assign <wire_name> = <expression>

 always <statement>
   ⋮
 always <statement>

endmodule
```

The order in which the function declarations, continuous assignments and always blocks are listed is not significant.

For simplicity, SV0 has no explicit variable declarations. A variable is a wire if it occurs on the left hand side of a continuous assignment, otherwise it is a register. Wires are ranged over by the syntactic meta-variable $\mathcal{W}$, registers are ranged over by $\mathcal{R}$ and both wires and registers are ranged over by $\mathcal{V}$. Details of Verilog's datatypes (e.g. bit widths) are ignored in SV0.

The results of functions are returned by an assignment to the function name inside its body. Thus a function name is also a register name.

A port is an output port if it is a wire and occurs on the left hand side of a continuous assignment or is a register and occurs on the left of a (blocking or non-blocking) procedural assignment. Ports that are not output ports are input ports.

In the BNF that follows, constructs enclosed between curley braces { and } are optional.

1.1 Expressions

The structure of expressions is not elaborated in detail for SV0.

It is assumed that wires and registers are expressions and that there is an operation of substituting an expression $\mathcal{E}_1$ for a variable $\mathcal{V}$ (which can be either a wire or a register) in another expression $\mathcal{E}_2$. This is denoted by $\mathcal{E}_2[\mathcal{V} \leftarrow \mathcal{E}_1]$. Note that in standard Verilog such substitution is not always possible. For example, `r[0]` is legitimate, but substituting `s+t` for `r` results in the illegal expression `(s+t)[0]`.

For the purpose of giving examples, the normal expression syntax of Verilog will be used.

1.2 Module items

Module items $\mathcal{I}$ in SV0 are constructed from expressions (ranged over by $\mathcal{E}$), event expressions (ranged over by $\mathcal{T}$) and statements (ranged over by $\mathcal{S}$).

$$
\begin{array}{lcll}
\mathcal{I} & ::= & \texttt{function}\ \mathcal{F}\texttt{;} & \text{(Function declaration)} \\
 & & \quad \texttt{input}\ \mathcal{V}_1\texttt{;}\ \ldots\ \mathcal{V}_n\texttt{;} & \\
 & & \quad \mathcal{S} & \\
 & & \texttt{endfunction} & \\
 & | & \texttt{assign}\ \mathcal{W}\ \texttt{=}\ \mathcal{E} & \text{(Continuous assignment)} \\
 & | & \texttt{always}\ \mathcal{S} & \text{(Always block)}
\end{array}
$$

The bodies of functions are not allowed to contain timing controls (see 1.3).

1.3 Event expressions

Event expressions $\mathcal{T}$ only occur as components of timing controls `@(`$\mathcal{T}$`)`. They can be used both to delimit synchronous cycle boundaries and to specify combinational logic. Only the following kinds of event expressions are allowed in SV0:

$$
\begin{array}{lcll}
\mathcal{T} & ::= & \mathcal{V} & \text{(Change of value)} \\
 & \mid & \texttt{posedge}\ \mathcal{V} & \text{(Positive edge)} \\
 & \mid & \texttt{negedge}\ \mathcal{V} & \text{(Negative edge)} \\
 & \mid & \mathcal{T}_1\ \texttt{or}\ \cdots\ \texttt{or}\ \mathcal{T}_n & \text{(Compound sensitivity list)}
\end{array}
$$

1.4 Statements

The syntax of statements $\mathcal{S}$ is given by the BNF below. The variables $\mathcal{R}$ and $\mathcal{B}$ range over register names and block names, respectively; n ranges over positive numbers.

$$
\begin{array}{lcll}
\mathcal{S} & ::= & \texttt{()} & \text{(Empty statement)} \\
 & \mid & \mathcal{R}\ \texttt{=}\ \mathcal{E} & \text{(Blocking assignment)} \\
 & \mid & \mathcal{R}\ \texttt{<=}\ \mathcal{E} & \text{(Non-blocking assignment)} \\
 & \mid & \texttt{begin}\{\texttt{:}\mathcal{B}\}\ \ \mathcal{S}_1\texttt{;}\ \cdots\ \texttt{;}\ \mathcal{S}_{\mathrm{n}}\ \texttt{end} & \text{(Sequencing block)} \\
 & \mid & \texttt{disable}\ \mathcal{B} & \text{(Disable statement)} \\
 & \mid & \texttt{if}\ (\mathcal{E})\ \mathcal{S}_1\ \{\texttt{else}\ \mathcal{S}_2\} & \text{(Conditional)} \\
 & \mid & \texttt{case}\ (\mathcal{E}) & \text{(Case statement)} \\
 & & \quad \mathcal{E}_1\texttt{:}\ \mathcal{S}_1 & \\
 & & \quad\quad \vdots & \\
 & & \quad \mathcal{E}_n\texttt{:}\ \mathcal{S}_n & \\
 & & \quad \{\texttt{default:}\ \mathcal{S}_{n+1}\} & \\
 & & \texttt{endcase} & \\
 & \mid & \texttt{while}\ (\mathcal{E})\ \mathcal{S} & \text{(While-statement)} \\
 & \mid & \texttt{repeat}\ (n)\ \mathcal{S} & \text{(Repeat statement)} \\
 & \mid & \texttt{for}\ (\mathcal{R}_1\texttt{=}\mathcal{E}_1\texttt{;}\ \mathcal{E}\texttt{;}\ \mathcal{R}_2\texttt{=}\mathcal{E}_2)\ \mathcal{S} & \text{(For statement)} \\
 & \mid & \texttt{forever}\ \mathcal{S} & \text{(Forever-statement)} \\
 & \mid & \texttt{@(}\mathcal{T}\texttt{)}\ \mathcal{S} & \text{(Timing control)}
\end{array}
$$

The following syntactic restrictions are assumed in SV0:

1. Each register can be assigned to in at most one always block.
2. Every disable statement `disable` $\mathcal{B}$ occurs inside a sequential block `begin:`$\mathcal{B} \cdots$ `end`.
3. Every path through the body of a while, forever or for statement must contain a timing control. This is checked by the symbolic exection algorithm in 2.3.

Other restrictions will be needed to ensure that the cycle semantics is consistent with the event semantics.

Case-statements, repeat-statements and for-statements are regarded as abbreviations for combinations of other statements (see 2.2.3).

Chapter 2

Cycle Semantics

The semantics of a module is represented by a Mealy machine whose inputs are determined by the input ports of the module and whose outputs are determined by its output ports. The state vector of the machine consists of the registers written by assignments in each always block together with additional control registers, called program counters. Program counters will be named $\texttt{pc}, \texttt{pc}_1, \texttt{pc}_2, \texttt{pc}_3$ etc. – a separate one for each always block. In the initial state all program counters are assumed to be `0`, but the initial values of other components of the state (i.e. the registers) is not specified.

A purely combinational module will have exactly one state, so is equivalent to a function from a vector of input values to a vector of output values.

A (Mealy) machine will be represented textually by a set of equations describing combinational logic together with next state assertions. These will be written using a Verilog-like notation. Such a 'meta-circular' use of Verilog to describe itself is intended to be readable and informal. A more rigorous symbolic representation of Mealy machines inside a formal logic will be needed for equivalence checking.

An equation $\mathcal{W} = \mathcal{E}$ asserts that the value of $\mathcal{W}$ is equal to the value of expression $\mathcal{E}$. For example, the equation:

```
out = in1+in2
```

defines the combinational addition function.

Continuous assignments `assign` $\mathcal{W}$ `=` $\mathcal{E}$ are interpreted as equations $\mathcal{W}$ `=` $\mathcal{E}$.

A function declaration like

```
function 𝓕;
  input 𝓥₁; ... 𝓥ₙ;
  𝓢
 endfunction
```

generates an equation of the form

$\mathcal{F}(\mathcal{V}_1, \ldots, \mathcal{V}_n) \;=\; \mathcal{E},$

where $\mathcal{E}$ is obtained by symbolically executing the function body $\mathcal{S}$.

For example:

```
function f;
 input a, b, c, d;
 begin
  f = a;
  if (b)
   begin
    if (c) f = d; else f = !d;
   end
 end
```

generates the equation : $\mathtt{f}(\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}) \;=\;$ `b ? c ? d : !d : a`. How this equation is derived is explained later.

Always blocks generate a set of next-state assertions involving the registers in the block and a program counter (denoted by `pc` in the examples that follow).

Next-state assertions will be represented with Verilog-like phrases of the form

`@(`$\mathcal{T}$`) if (`$\mathcal{E}$`) begin` $\mathcal{R}_1$ `<=` $\mathcal{E}_1$; $\ldots$; $\mathcal{R}_n$ `<=` $\mathcal{E}_n$ `end`

which means that when $\mathcal{T}$ occurs and $\mathcal{E}$ is true, then the state is updated according to the listed assignments. Statements that perform assignments before the first timing control will generate an initialization not guarded by any `@(`$\mathcal{T}$`)` (see examples 7 and 8 in 2.1). This also happens for function bodies, which contain no timing controls.

2.1 Examples

The examples in this section are intended to give the idea of the semantics. A precise specification is given in 2.2 and 2.3.

Example 1

The example below sets `a` to `0` on the first edge and then sets `b` to `a` on the second edge. Thereafter `a` and `b` are updated with `0` on each cycle.

```
always @(posedge clk) begin a=0; @(posedge clk) b=a; end
```

generates two next-state assertions:

```
@(posedge clk)
   if (pc == 0)
    begin
       pc <= 1;
       a <= 0;
       b <= b;
    end

@(posedge clk)
   if (pc == 1)
    begin
       pc <= 0;
       a <= a;
       b <= a;
    end
```

Example 2

The following example is a state machine described in an implicit style. It is Example 8-16 from the Synopsys HDL Compiler for Verilog Reference Manual [5].

```
always
 begin
  @(posedge clk) total = data;
  @(posedge clk) total = total + data;
  @(posedge clk) total = total + data;
 end
```

which generates three next-state assertions:

```
@(posedge clk)
   if (pc == 0)
    begin
       pc <= 1;
       total <= data;
    end

@(posedge clk)
   if (pc == 1)
    begin
       pc <= 2;
       total <= (total) + data;
    end

@(posedge clk)
   if (pc == 2)
    begin
       pc <= 0;
       total <= (total) + data;
    end
```

Example 3

An explicit style of description of the machine in Example 2 is given next. This is Example 8-17 from the Synopsys HDL Compiler for Verilog Reference Manual [5].

```
always
 @(posedge clk)
 begin
  case (state)
  0:  begin total = data;
            state = 1;
      end
  1:  begin total = total + data;
            state = 2;
      end
  default:
      begin total = total + data;
            state = 0;
      end
  endcase
 end
```

This generates:

```
@(posedge clk)
   if (pc == 0)
    begin
       pc <= 0;
       total <= (state == 0) ? data : (total) + data;
       state <=  (state == 0) ? 1 : (state == 1) ? 2 : 0;
    end
```

Note that the program counter generated from the implicit state machine specification corresponds to the register `state` in the explicit state specification. The explicit states style of state machine specification makes the program counter `pc` redundant.

Example 4

Another example illustrating a redundant program counter is:

```
always @(posedge clk)
          if (p) begin a=b;  b=a;  end
          else   begin a<=b; b<=a; end
```

generates

```
@(posedge clk)
   if (pc == 0)
    begin
       pc <= 0;
       a <= b;
       b <= p ? b : a;
    end
```

Example 5

Asynchronous (combinational) always blocks also lead to a redundant program counter. For example:

```
always @(b or c) a = b + c
```

generates

```
@(b or c)
   if (pc == 0)
    begin
       pc <= 0;
       a <= b + c;
    end
```

Since whenever `b` and `c` change, `a` is updated, it follows (induction over time – details elsewhere) that this next-state assertion is equivalent to the equation `a = b+c`. However consider instead:

```
always @(b or c) if (p) a = b+c;
```

which generates:

```
@(b or c)
   if ( pc == 0 )
    begin
       pc <= 0;
       a <= p ? b + c : a;
    end
```

Suppose `a` equals `b+c`. If `b` or `c` then changes when `p` is false, then `a` will become different from `b+c`. Thus `a`'s value must be latched – hence the need for synthesizers to do latch inference.

Example 6

Here is a combinational example that doesn't lead to any latch inference.

```
always
 @(a or b or c or d)
 begin
  f = a;
  if (b)
   begin
    if (c) f = d; else f = !d;
   end
 end
```

generates:

```
@(a or b or c or d)
   if (pc == 0)
    begin
       pc <= 0;
       f <= b ? c ? d : !d : a;
    end
```

Example 7

The sequential block in Example 6, namely:

```
begin
 f = a;
 if (b)
  begin
   if (c) f = d; else f = !d;
  end
end
```

was the body of the example function named `f` given on page 6. This statement (without any `always` and timing control) generates:

```
if (pc ==0 )
 begin
    pc <= 1;
    f <= b ? c ? d : !d : a;
 end
```

The expression assigned to the function name `f` is used to generate the equation defining `f` (see page 22 at the end of 2.3).

Example 8

Each next-state assertion, except for any initialisation, is guarded by a separate timing control. This allows for the possibility (usually prohibited by synthesizers) that there may be different timing controls along different paths.

A (non-synthesizable) nonsense statement is used to illustrate this:

```
always if (p) begin
                 a=1;
                 @(posedge clk) b=2;
                 @(negedge clk) c=3;
              end
       else   begin
                 a=5;
                 @(clk) b=6;
              end
```

generates four next-state assertions (the first of which is an initialisation):

```
   if (pc == 0)
     begin
        pc <= p ? 1 : 3;
        c <= c;
        a <= p ? 1 : 5;
        b <= b;
     end

@(posedge clk)
   if (pc == 1)
    begin
       pc <= 2;
       c <= c;
       a <= a;
       b <= 2;
    end

@(negedge clk)
   if (pc == 2)
    begin
       pc <= p ? 1 : 3;
       c <= 3;
       a <= p ? 1 : 5;
       b <= b;
    end

@(clk)
   if (pc == 3)
    begin
       pc <= p ? 1 : 3;
       c <= c;
       a <= p ? 1 : 5;
       b <= 6;
    end
```

The machine represented by a complete module is obtained by combining (conjoining) the equations and next-state assertions generated by each function declaration, continuous assignment and always block (see 2.4).

Next-state equations are obtained by symbolically executing the result of translating $\mathcal{S}$ to a semantic pseudo-code.

2.2 Semantic Pseudo-Code

The semantics of SV0 is given in two stages. First, all statements are converted to a semantic pseudo-code. This reduces Verilog's sequential control flow constructs to a simple uniform form. Second the pseudo-code is interpreted. For synthesizable Verilog, a cycle based interpretation is appropriate, however the semantic pseudo-code is also a suitable vehicle for giving an event based semantics [2].

It is hoped that a common pseudo-code can be developed to provide a 'deep structure' for both Verilog and VHDL, thus reducing the differences between the two languages to just 'surface structure'.

2.2.1 Pseudo-code instructions

Statements are compiled to pseudo-code consisting of sequences of instructions from the following instruction set:

$\mathcal{R}$ `=` $\mathcal{E}$	blocking assignment
$\mathcal{R}$ `<=` $\mathcal{E}$	non-blocking assignment
`@(`$\mathcal{T}$`)`	timing control
`go` n	unconditional jump to instruction n
`ifnot` $\mathcal{E}$ `go` n	jump to instruction n if $\mathcal{E}$ is not true
`disable` $\mathcal{B}$	disable (break out of) block $\mathcal{B}$

2.2.2 Example translations

Before giving the straightforward algorithm for translating from SV0 statement to pseudo-code, some example translations are presented.

Example 1

```
if (ℰ)
 begin a<=b; b<=a; end
else
 begin a=b; b=a; end
```

translates to:

```
0:   ifnot ℰ go 4
1:   a <= b
2:   b <= a
3:   go 6
4:   a = b
5:   b = a
```

Example 2

```
if (ℰ)
 begin a<=b; @(posedge clk) b<=a; end
else
 begin a=b; b=a; end
```

translates to

```
0:   ifnot ℰ go 5
1:   a <= b
2:   @(posedge clk)
3:   b <= a
4:   go 7
5:   a = b
6:   b = a
```

Example 3

```
if (ℰ)
 begin a<=b; @(posedge clk) b<=a; end
else
 begin a=b; @(posedge clk) b=a; end
```

translates to

```
0:   ifnot ℰ go 5
1:   a <= b
2:   @(posedge clk)
3:   b <= a
4:   go 8
5:   a = b
6:   @(posedge clk)
7:   b = a
```

Example 4

```
if (ℰ)
 begin:b1 a<=b; disable b1; b<=a; end
else
 begin a=b; @(posedge clk) b=a; end
```

translates to

```
0:   ifnot ℰ go 5
1:   a <= b
2:   go 4
3:   b <= a
4:   go 8
5:   a = b
6:   @(posedge clk)
7:   b = a
```

Example 5

```
forever @(b or c) a = b + c;
```

translates to

```
0:   @(b or c)
1:   a = b + c
2:   go 0
```

Example 6

```
forever
 begin
  @(posedge clk) total = data;
  @(posedge clk) total = total + data;
  @(posedge clk) total = total + data;
 end
```

translates to

```
0:   @(posedge clk)
1:   total= data
2:   @(posedge clk)
3:   total = total + data)
4:   @(posedge clk)
5:   total = total + data
6:   go 0
```

Example 7

```
forever
 @(posedge clk)
 begin
  case (state)
  0:  begin total = data;
            state = 1;
      end
  1:  begin total = total + data;
            state = 2;
      end
  default:
      begin total = total + data;
            state = 0;
      end
  endcase
 end
```

translates to

```
0:   @(posedge clk)
1:   ifnot state == 0 go 5
2:   total = data
3:   state= 1
4:   go 11
5:   ifnot state == 1 go 9
6:   total = total + data
7:   state = 2
8:   go 11
9:   total = total + data
10:  state = 0
11:  go 0
```

2.2.3 Macro-expansion of derived constructs

The first step in translating statements to pseudo-code is to 'macro-expand' case, repeat and for statements.

Case statements

$$
\begin{array}{l}
\texttt{case}\ (\mathcal{E}) \\
\quad \mathcal{E}_1\texttt{:}\ \mathcal{S}_1 \\
\quad \mathcal{E}_2\texttt{:}\ \mathcal{S}_2 \\
\quad\ \ \vdots \\
\quad \mathcal{E}_n\texttt{:}\ \mathcal{S}_n \\
\quad \{\texttt{default:}\ \mathcal{S}_{n+1}\} \\
\texttt{endcase}
\end{array}
$$

is expanded to:

$\texttt{if (}\mathcal{E}\texttt{==}\mathcal{E}_1\texttt{)}\ \mathcal{S}_1\ \texttt{else if (}\mathcal{E}\texttt{==}\mathcal{E}_2\texttt{)}\ \mathcal{S}_2\ \cdots\ \texttt{else if (}\mathcal{E}\texttt{==}\mathcal{E}_n\texttt{)}\ \mathcal{S}_n\ \{\texttt{else}\ \mathcal{S}_{n+1}\}$

Repeat statements

$\texttt{repeat (n)}\ \mathcal{S}$

is expanded to:

$$\texttt{begin}\ \underbrace{\mathcal{S}\texttt{;}\ \ldots\ \texttt{;}\mathcal{S}}_{n \text{ copies of } \mathcal{S}}\ \texttt{end}$$

For statements

$\texttt{for (}\mathcal{R}_1\texttt{=}\mathcal{E}_1\texttt{; }\mathcal{E}\texttt{; }\mathcal{R}_2\texttt{=}\mathcal{E}_2\texttt{) }\mathcal{S}$

is expanded to:

$\texttt{begin }\mathcal{R}_1\texttt{=}\mathcal{E}_1\texttt{; while (}\mathcal{E}\texttt{) begin }\mathcal{S}\texttt{; }\mathcal{R}_2\texttt{=}\mathcal{E}_2\texttt{ end end}$

2.2.4 The size of a statement

The size function defined in this section is used in the translation algorithm described in 2.2.5. Let the size $|\mathcal{S}|$ of $\mathcal{S}$ be as defined below inductively on the structure of $\mathcal{S}$. It will turn out that $|\mathcal{S}|$ is the number of instructions that $\mathcal{S}$ is translated to.

$$\begin{array}{lcl}
|\mathcal{R}\ \texttt{=}\ \mathcal{E}| & = & 1 \\
|\mathcal{R}\ \texttt{<=}\ \mathcal{E}| & = & 1 \\
|\texttt{begin}\{\texttt{:}\mathcal{B}\}\ \texttt{end}| & = & 0 \\
|\texttt{begin}\{\texttt{:}\mathcal{B}\}\ \ \mathcal{S}_1\texttt{;}\cdots\texttt{;}\ \mathcal{S}_{\rm n}\ \texttt{end}| & = & |\mathcal{S}_1| + \ \cdots\ + |\mathcal{S}_n| \\
|\texttt{disable}\ \mathcal{B}| & = & 1 \\
|\texttt{if (}\mathcal{E}\texttt{) }\mathcal{S}| & = & |\mathcal{S}| + 1 \\
|\texttt{if (}\mathcal{E}\texttt{) }\mathcal{S}_1\texttt{ else }\mathcal{S}_2| & = & |\mathcal{S}_1| + |\mathcal{S}_2| + 2 \\
|\texttt{while (}\mathcal{E}\texttt{) }\mathcal{S}| & = & |\mathcal{S}| + 2 \\
|\texttt{forever }\mathcal{S}| & = & |\mathcal{S}| + 1 \\
|\texttt{@(}\mathcal{T}\texttt{)}| & = & 1
\end{array}$$

The size of a sequence of statements is defined to be the sum of the sizes of the components of the sequence. Thus if $\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ is a sequence of statements, then define:

$$\begin{array}{lcl}
|\langle\rangle| & = & 0 \\
|\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle| & = & |\mathcal{S}_1| + \ \cdots\ + |\mathcal{S}_n|
\end{array}$$

2.2.5 Translation algorithm

The sequence $\langle \mathtt{i}_0, \ldots, \mathtt{i}_n \rangle$ of instructions that statement $\mathcal{S}$ is translated to is denoted by $[\![\mathcal{S}]\!]\ p$, where p is the position of the first instruction (e.g. go p jumps to the start of the program).

To handle sequential blocks, it is convenient to define in parallel the translation of a sequence $\langle \mathcal{S}_1, \ldots, \mathcal{S}_N \rangle$ of statements (see the third and forth clauses of the definition below).

In the definition below $\frown$ is sequence concatenation and $s[u \leftarrow v]$ denotes the result of replacing all occurrences of u in s by v.

$$
\begin{array}{lcl}
[\![\mathcal{R}\ \mathtt{=}\ \mathcal{E}]\!]\ p & = & \langle \mathcal{R}\ \mathtt{=}\ \mathcal{E} \rangle \\
[\![\mathcal{R}\ \mathtt{<=}\ \mathcal{E}]\!]\ p & = & \langle \mathcal{R}\ \mathtt{<=}\ \mathcal{E} \rangle \\
[\![\langle\rangle]\!]\ p & = & \langle\rangle \\
[\![\langle \mathcal{S}_1,\ \mathcal{S}_2, \ldots, \mathcal{S}_n \rangle]\!]\ p & = & [\![\mathcal{S}_1]\!]\ p \frown [\![\langle \mathcal{S}_2, \ldots, \mathcal{S}_n \rangle]\!](p{+}|\mathcal{S}_1|) \\
[\![\mathtt{begin}\{\mathtt{:}\mathcal{B}\}\ \ \mathcal{S}_1\mathtt{;} \cdots \mathtt{;}\ \mathcal{S}_\mathrm{n}\ \mathtt{end}]\!]\ \mathrm{p} & = & [\![\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle]\!]\ p\ [\mathtt{disable}\ \mathcal{B} \leftarrow \mathtt{go}\ p{+}|\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle|] \\
[\![\mathtt{disable}\ \mathcal{B}]\!]\ p & = & \langle \mathtt{disable}\ \mathcal{B} \rangle \\
[\![\mathtt{if}\ \mathtt{(}\mathcal{E}\mathtt{)}\ \mathcal{S}]\!]\ p & = & \langle \mathtt{ifnot}\ \mathcal{E}\ \mathtt{go}\ p{+}|\mathcal{S}|{+}1 \rangle \frown [\![\mathcal{S}]\!](p+1) \\
[\![\mathtt{if}\ \mathtt{(}\mathcal{E}\mathtt{)}\ \mathcal{S}_1\ \mathtt{else}\ \mathcal{S}_2]\!]\ p & = & \langle \mathtt{ifnot}\ \mathcal{E}\ \mathtt{go}\ p{+}|\mathcal{S}_1|{+}2 \rangle \\
 & & \quad \frown [\![\mathcal{S}_1]\!](p{+}1) \\
 & & \quad \frown \langle \mathtt{go}\ p{+}|\mathcal{S}_1|{+}|\mathcal{S}_2|{+}2 \rangle \\
 & & \quad \frown [\![\mathcal{S}_2]\!](p{+}|\mathcal{S}_1|{+}2) \\
[\![\mathtt{while}\ \mathtt{(}\mathcal{E}\mathtt{)}\ \mathcal{S}]\!]\ p & = & \langle \mathtt{ifnot}\ \mathcal{E}\ \mathtt{go}\ p{+}|\mathcal{S}|{+}2 \rangle \frown [\![\mathcal{S}]\!](p{+}1) \frown \langle \mathtt{go}\ p \rangle \\
[\![\mathtt{forever}\ \mathcal{S}]\!]\ p & = & [\![\mathcal{S}]\!]\ p \frown \langle \mathtt{go}\ p \rangle \\
[\![\mathtt{@(}\mathcal{T}\mathtt{)}\ \mathcal{S}]\!]\ p & = & \langle \mathtt{@(}\mathcal{T}\mathtt{)} \rangle \frown [\![\mathcal{S}]\!](p{+}1)
\end{array}
$$

2.3 From pseudo-code to next-state assertions

Next-state assertions are generated from the pseudo-code by symbolic execution until a timing control is reached. When a conditional jump is encountered, both paths are followed and then the results combined.

As pseudo-code is symbolically executed, blocking assignments are performed on a symbolic representation of the state, but non-blocking assignments are accumulated and only performed at the end of the cycle – i.e. when a timing control is reached.

A symbolic state is represented by a set of pairs associating registers with expressions (i.e. a finite function). The following notation is used:

$$\{\mathcal{R}_1 \mapsto \mathcal{E}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{E}_n\}$$

This denotes a state in which register $\mathcal{R}_i$ has the value $\mathcal{E}_i$ $(1 \leq i \leq n)$.

A special control register called the program counter is assumed. Different always blocks in a module are assumed to have different program counters, which will be named $\mathtt{pc}, \mathtt{pc}_1, \mathtt{pc}_2, \mathtt{pc}_3$ etc.

The accumulating set of pending non-blocking assignments will be denoted by:

$$\{\mathcal{R}_1\texttt{<=}\mathcal{E}_1, \ldots, \mathcal{R}_n\texttt{<=}\mathcal{E}_n\}$$

The symbolic execution algorithm starts at a given instruction and then steps through the pseudo-code, updating the state and pending non-blocking assignments until a timing control is reached. The pending assignments are then performed.

Programs whose symbolic execution generates an infinite loop can result from while-statements that have a path through their body that is not broken by a timing control. Such statements are excluded from SV0.

Recall that the instruction set is:

$\mathcal{R}$ `=` $\mathcal{E}$	blocking assignment
$\mathcal{R}$ `<=` $\mathcal{E}$	non-blocking assignment
`@(`$\mathcal{T}$`)`	timing control
`go` n	unconditional jump to instruction n
`ifnot` $\mathcal{E}$ `go` n	jump to instruction n if $\mathcal{E}$ is not true
`disable` $\mathcal{B}$	disable (break out of) block $\mathcal{B}$

The result of simultaneously (i.e. in parallel) substituting the expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$ for the registers $\mathcal{R}_1, \ldots, \mathcal{R}_n$ in an expression $\mathcal{E}$ is denoted by:

$$\mathcal{E}[\mathcal{R}_1, \ldots, \mathcal{R}_n \leftarrow \mathcal{E}_1, \ldots, \mathcal{E}_n]$$

The symbolic execution algorithm takes a state and a set of pending non-blocking assignments and returns a state.

The 'current instruction' is the one pointed to by the program counter.

The symbolic execution algorithm is as follows.

1. If $\texttt{pc} \mapsto i$ and instruction i is $\mathcal{R}$ `=` $\mathcal{E}$ then:
   - let $\mathcal{E}' = \mathcal{E}[\mathcal{R}_1, \ldots, \mathcal{R}_n \leftarrow \mathcal{E}_1, \ldots, \mathcal{E}_n]$ (so $\mathcal{E}'$ is the value of $\mathcal{E}$ in the current state);
   - if the state doesn't contain any assignment to $\mathcal{R}$, then extend the state with $\mathcal{R} \mapsto \mathcal{E}'$;
   - if the state contains an assignment to $\mathcal{R}$ (e.g. $\mathcal{R} \mapsto \mathcal{R}_i$, for some i) then replace this assignment with $\mathcal{R} \mapsto \mathcal{E}'$;
   - increment the program counter so that $\texttt{pc} \mapsto i + 1$;
   - recursively invoke symbolic execution with the modified state and the same pending non-blocking assignments.
2. If $\texttt{pc} \mapsto i$ and instruction i is $\mathcal{R}$ `<=` $\mathcal{E}$ then:
   - let $\mathcal{E}' = \mathcal{E}[\mathcal{R}_1, \ldots, \mathcal{R}_n \leftarrow \mathcal{E}_1, \ldots, \mathcal{E}_n]$
   - if the set of pending non-blocking assignments doesn't contain any assignment to $\mathcal{R}$, then extend the set with $\mathcal{R}$ `<=` $\mathcal{E}'$;
   - if the pending non-blocking assignments contains an assignment to $\mathcal{R}$ then replace this assignment with $\mathcal{R}$ `<=` $\mathcal{E}'$ (thus later non-blocking assignments override earlier ones to the same variable);
   - increment the program counter so that $\texttt{pc} \mapsto i + 1$;
   - recursively invoke symbolic execution with the modified state and the extended list of pending non-blocking assignments.
3. If $\texttt{pc} \mapsto i$ and instruction i is a timing control, or if i points outside the program, then perform the pending non-blocking assignments (overriding any assignments in the state, if necessary) and return the resulting state. This state consists of $\texttt{pc} \mapsto i + 1$ and those $\mathcal{R}_i \mapsto \mathcal{E}_i$ in the symbolic state for which there is no pending non-blocking assignment to $\mathcal{R}_i$ together with all $\mathcal{R} \mapsto \mathcal{E}$ where $\mathcal{R}$ `<=` $\mathcal{E}$ is a pending non-blocking assignment.
4. If $\texttt{pc} \mapsto i$ and instruction i is `go` n then set `pc` to n and recursively invoke symbolic execution with the modified state and the same pending non-blocking assignments.

5. If $\texttt{pc} \mapsto i$ and instruction i is `ifnot` $\mathcal{E}$ `go` n then:
   - let $\mathcal{E}' = \mathcal{E}[\mathcal{R}_1, \ldots, \mathcal{R}_n \leftarrow \mathcal{E}_1, \ldots, \mathcal{E}_n]$
   - let $\{\texttt{pc} \mapsto j,\ \mathcal{R}_1 \mapsto \mathcal{E}^f_1,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}^f_n\}$ be the state resulting from recursively symbolically executing with $\texttt{pc} \mapsto n$;
   - let $\{\texttt{pc} \mapsto k,\ \mathcal{R}_1 \mapsto \mathcal{E}^t_1,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}^t_n\}$ be the state resulting from recursively symbolically executing with $\texttt{pc} \mapsto i+1$;
   - return as the result of the symbolic execution the state $\{\texttt{pc} \mapsto \mathcal{E}'\ ?\ k\ :\ j,\ \mathcal{R}_1 \mapsto \mathcal{E}'\ ?\ \mathcal{E}'^t_1\ :\ \mathcal{E}'^f_1,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}'\ ?\ \mathcal{E}'^t_n\ :\ \mathcal{E}'^f_n\}$
6. The instruction `disable` $\mathcal{B}$ should not be generated. SV0 assumes that only an enclosing block can be disabled and all such disables are replaced by jumps during the compilation of sequential blocks.

The symbolic execution algorithm given above is used to generate next-state assertions from a statement as follows.

If the first instruction is not a timing control, then generate an initialization assertion:

$$\texttt{if (pc == 0) begin pc <= } j;\ \mathcal{R}_1\ \texttt{<=}\ \mathcal{E}_1;\ \ldots\ ;\ \mathcal{R}_n\ \texttt{<=}\ \mathcal{E}_n;\ \texttt{end}$$

where $\{\texttt{pc} \mapsto j, \mathcal{R}_1 \mapsto \mathcal{E}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{E}_n\}$ is the state resulting from symbolic execution starting with $\{\texttt{pc} \mapsto 0, \mathcal{R}_1 \mapsto \mathcal{R}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{R}_n\}$ and the empty set of pending non-blocking assignments.

Next, for each value i of the program counter that points to a timing control instruction $\texttt{@}(\mathcal{T})$ generate an assertion

$$\texttt{@}(\mathcal{T})\ \texttt{if (pc == } i\texttt{) begin pc <= } j;\ \mathcal{R}_1\ \texttt{<=}\ \mathcal{E}_1;\ \ldots\ ;\ \mathcal{R}_n\ \texttt{<=}\ \mathcal{E}_n;\ \texttt{end}$$

where $\{\texttt{pc} \mapsto j, \mathcal{R}_1 \mapsto \mathcal{E}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{E}_n\}$ is the state resulting from symbolic execution starting with $\{\texttt{pc} \mapsto i, \mathcal{R}_1 \mapsto \mathcal{R}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{R}_n\}$ and the empty set of pending non-blocking assignments.

The next-state assertions from an always block `always` $\mathcal{S}$ are obtained by generating the assertions from the statement `forever` $\mathcal{S}$.

The equation generated by a function defined by:

```
function F;
  input V1; ... Vn;
  S
endfunction
```

is obtained by generating the assertions from the body $\mathcal{S}$. If the function is well-formed there should only be one next-state assertion of the form:

$$
\begin{array}{l}
\texttt{if (pc == 0)} \\
\ \texttt{begin} \\
\quad\ \texttt{pc <= 1;} \\
\qquad \vdots \\
\quad\ \mathcal{F}\ \texttt{<=}\ \mathcal{E}\texttt{;} \\
\qquad \vdots \\
\ \texttt{end}
\end{array}
$$

The equation defining $\mathcal{F}$ is then: $\mathcal{F}(\mathcal{V}_1, \ldots, \mathcal{V}_n) \ = \ \mathcal{E}$

2.4 The meaning of a module

The representation of the Mealy machine generated from a module:

$$
\begin{array}{l}
\texttt{module}\ \mathcal{M}\ \texttt{(}\mathcal{V}_1\texttt{,}\ \ldots\ \texttt{,}\mathcal{V}_q\texttt{);} \\
\ \texttt{function}\ \mathcal{F}_1\texttt{; input}\ \mathcal{V}_1^1\texttt{,}\ \ldots\ \texttt{,}\mathcal{V}_1^{i_1}\texttt{;}\ \mathcal{S}_{\mathcal{F}_1}\ \texttt{endfunction} \\
\quad \vdots \\
\ \texttt{function}\ \mathcal{F}_r\texttt{; input}\ \mathcal{V}_r^1\texttt{,}\ \ldots\ \texttt{,}\mathcal{V}_r^{i_r}\texttt{;}\ \mathcal{S}_{\mathcal{F}_r}\ \texttt{endfunction} \\
\ \texttt{assign}\ \mathcal{W}_1\ \texttt{=}\ \mathcal{E}_1 \\
\quad \vdots \\
\ \texttt{assign}\ \mathcal{W}_s\ \texttt{=}\ \mathcal{E}_s \\
\ \texttt{always}\ \mathcal{S}_1 \\
\quad \vdots \\
\ \texttt{always}\ \mathcal{S}_t \\
\texttt{endmodule}
\end{array}
$$

consists of:

1. an equation $\mathcal{F}_j(\mathcal{V}_j^1, \ldots, \mathcal{V}_j^{i_j}) \ = \ \mathcal{E}_j$ for each function $(1 \leq j \leq r)$;
2. an equation $\mathcal{W}_j \ = \ \mathcal{E}_j$ for each continuous assignment $(1 \leq j \leq s)$;
3. the union of the assertions generated by each always block, each one with a different program counter, say $\texttt{pc}_j$ $(i \leq j \leq t)$.

Bibliography

[1] Richard Boulton. The Computer Language Reasoning Tool. See www.dai.ed.ac.uk/daidb/staff/personal_pages/rjb/claret/index.html.

[2] M. J. C. Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 136–145. IEEE Computer Society Press, 1995.

[3] David Greaves. The *CSYN* Verilog compiler and other tools. Available from www.cl.cam.ac.uk/users/djg/localtools/index.html.

[4] Daryl Stewart. The Verilog Formal Equivalence Project. Available from www.cl.cam.ac.uk:80/users/djs1002/verilog.project/syntax/.

[5] Synopsys, Inc. *HDL Compiler for Verilog Reference Manual, Version 3.5*, September 1996.

[6] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 3rd edition, 1996.