Synthesizable Verilog
syntax and semantics

Mike Gordon
VFE Project
University of Cambridge Computer Laboratory

Version 0.7

June 4, 1997

WARNING: this is a working draft containing preliminary material, some of which the reader is likely to find obscure.

The Verilog Formal Equivalence (VFE) Project is funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC). The Principal Investigator is Dr. David Greaves.

Contents

Preface

Synthesizable Verilog is a subset of the full Verilog HDL [9] that lies within the domain of current synthesis tools (both RTL and behavioral).

This document specifies a subset of Verilog called V0.[1] 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 – V0 – 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 (V1, V2 etc.) that will converge to the version of Verilog used in the VFE project[2] 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 clocked register transfer level (RTL) 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. The normal semantics of Verilog is based on events, i.e. changes in the values on wires and in registers. Such *event semantics* can accurately model detailed asynchronous behavour, but are very fine-grained and do not easily support formal verification. Most practical formal methods (e.g. model checking and theorem proving) are oriented towards descriptions of systems in terms of their execution traces, which are

[1] To avoid confustion with the Synchronous Verilog (SV) developed by Dr Ching-Tsun Chou at Fujitsu [2], the subset SV0 in Version 0.1 of this document has been renamed V0 (similarly for V1, V2 etc).

[2] 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*.

sequences (or trees) of states. One might characterise simulation semantics as 'edge-oriented' and trace semantics as 'level-oriented'. The relationship between the two views is obtained by accumulating the changes (events) during a simulation cycle to obtain the state holding at the end of the cycle. The sequence of states that the simulation cycle quiesces to at successive instants of simulation time will be called *simulation trace semantics* or just *trace semantics*. If there are race conditions, then there may be several possible successor states to a given state, so a tree will be needed to accurately characterise the event semantics (i.e. branching time). However, standard hardware synthesis methods create deterministic sequential machines whose executions can be characterised by linear traces. The trace semantics given to Verilog here will thus consist of sequences not trees. Part of our goal is to provide sufficient syntactic conditions to guarantee the linear trace semantics equivalent to the event semantics. Hardware synthesized from Verilog meeting these conditions will simulate the same as the source.

The trace semantics has the same timescale as the event (simulation) semantics – namely simulation time – but abstracts away from the individual events within a single simulation cycle (delta-time). Clocked sequential systems can also be viewed more abstractly in terms of the sequence of states held in registers during successive clock cycles. This view will be called the *clock cycle semantics* or just *cycle semantics*.[3] Certain kinds of hardware (e.g. transparent level sensitive latches) are rather badly approximated if only the states latched at clock edges are considered, so equivalences between such hardware is best done with trace semantics.

In the VFE project, fine-grain equivalence will be formulated in terms of trace semantics and a coarser grain (RTL) equivalence in terms of the cycle semantics. It is also intended to look at even looser equivalences at the behavioural level, in which operations can be moved across sequences of clock cyles (e.g. within the same 'super state').

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 lan-

[3] In earlier versions of this document the term "cycle semantics" was used confusingly to sometimes mean the trace semantics and sometimes the clock cycle semantics. We are having some difficultly converging on a good terminology, and further changes might occur.

guage constructs.

Some of the features in synthesizable Verilog missed out of `V0` 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 provides a simpler representation on which to define semantics. 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).

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 Chapter 4 is based on the algorithm underlying David Greaves' CSYN compiler [4]. The examples here were generated using a program built on top of Daryl Stewart's P1364 Verilog parser and pretty-printer [7] which is implemented using the syntax processing facilities of Richard Boulton's CLaReT system [1]. Thanks to Ching-Tsun Chou, Abhijit Ghosh, Daryl Stewart, Myra Van Inwegen and Steve Johnson for comments on a first draft of this document.

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 V0 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, V0 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 V0.

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 the curley braces "{" and "}" are optional.

1.1 Expressions

The structure of expressions is not elaborated in detail for V0.

It is assumed that there is an 'undefined' expression `x` (`1'bx` in Verilog), 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 V0 are constructed from expressions (ranged over by $\mathcal{E}$), event expressions (ranged over by $\mathcal{T}$) and statements (ranged over by $\mathcal{S}$).

$\mathcal{I}$	::=	`function` $\mathcal{F}$`;`
		  `input` $\mathcal{V}_1$`;` ... $\mathcal{V}_n$`;`
		  $\mathcal{S}$
		`endfunction`
			(Function declaration)

	|	`assign` $\mathcal{W}$ `=` $\mathcal{E}$	(Continuous assignment)
	|	`always` $\mathcal{S}$	(Always block)

The bodies of functions are not allowed to contain event expressions (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 cycle boundaries and to specify combinational logic. Only the following kinds of event expressions are allowed in V0:

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

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.

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

The following syntactic restrictions are assumed in V0:

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 4.2.

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 2.3).

Chapter 2

Semantic Pseudo-Code

The semantics of `V0` 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. A simplified event semantics in given Chapter 3, a trace semantics in Chapter 4 and a cycle semantics in Chapter 5.

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'.

2.1 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 Example translations

Before giving the straightforward algorithm for translating from `V0` statement to pseudo-code, some example translations are presented.

2.2.1

```
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
```

2.2.2

```
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
```

2.2.3

```
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
```

2.2.4

```
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
```

2.2.5

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

translates to

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

2.2.6

```
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
```

2.2.7

```
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
```

2.3 Macro-expansion of derived constructs

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

2.3.1 Case statements

`case` $(\mathcal{E})$
  $\mathcal{E}_1$`:` $\mathcal{S}_1$
  $\mathcal{E}_2$`:` $\mathcal{S}_2$
  $\vdots$
  $\mathcal{E}_n$`:` $\mathcal{S}_n$
  $\{$`default:` $\mathcal{S}_{n+1}\}$
`endcase`

is expanded to:

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

2.3.2 Repeat statements

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

is expanded to:

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

2.3.3 For statements

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

is expanded to:

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

2.4 The size of a statement

The size function defined in this section is used in the translation algorithm described in 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.

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

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:

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

2.5 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}
$$

Chapter 3

Event Semantics

The event semantics given here is a highly simplified version of the the semantics of the V language [3]. V has been used by Daryl Stewart as the basis for an accurate simulation semantics of most of the behavioral constructs of P1364 Verilog [5] and by David Greaves as the basis of his CSIM simulator. V0 only requires a simplified semantics because it has no delay controls (#n).

With an event semantics each always block and continuous assignment is represented by a concurrently running thread. The simulation cycle then non-deterministically executes threads according to the timing controls present.

It is assumed that each input port is driven by the environment with a sequence of values: the elements of the sequence being the values at successive instants of simulation time. When time advances, the values being input at the new time may change from their previous value. This change may cause an event expression $\mathcal{T}$ to 'fire' and any threads waiting on @($\mathcal{T}$) will then become enabled. The simulation cycle consists of repeatedly choosing an enabled thread, executing the next instruction in it, and then enabling any new threads that are waiting on timing controls that fire. If several threads are simultaneously enabled, then the choice of which thread to advance is non-deterministic. If there are no more enabled threads, pending non-blocking assignments are performed and further threads may become enabled and the cycle continues. If the cycle ever quiesces (i.e. no enabled thread and all pending non-blocking assignments processed), then time is advanced, the next set of inputs is considered and the whole process repeats.

Thus sequences of values on the input ports non-deterministically generate sequences on the output ports.

In V0, functions are eliminated by 'inlining' them using the equation generated by symbolic evaluation. Thus each function call $\mathcal{F}(\mathcal{E}_1,\ldots,\mathcal{E}_n)$ is replaced by $\mathcal{E}[\mathcal{V}_1,\ldots,\mathcal{V}_n \leftarrow \mathcal{E}_1,\ldots,\mathcal{E}_n]$ where $\mathcal{F}_j(\mathcal{V}_j^1,\ldots,\mathcal{V}_j^{i_j}) = \mathcal{E}_j$ is the equation generated from the declaration of $\mathcal{F}$, as described at the end of 4.2.

In `V0`, there is no semantic difference between wires and registers and a continuous assignment `assign` $\mathcal{W}$ `=` $\mathcal{E}$ is considered to be the always block `always @(`$\mathcal{T}_\mathcal{E}$`)` $\mathcal{W}$ `=` $\mathcal{E}$ where $\mathcal{T}_\mathcal{E}$ is $\mathcal{V}_1$ `or` $\cdots$ `or` $\mathcal{V}_n$ ($\mathcal{V}_1$, ..., $\mathcal{V}_n$ being the variables occurring in $\mathcal{E}$).

After inlining functions and converting continuous assignments to always blocks, a module can be considered to consist of a set of blocks `always` $\mathcal{S}_i$. The event semantics is specified by compiling `forever` $\mathcal{S}_i$ to pseudo code, for each i. The execution of the resulting pseudo code programs give rise to a separate simulation (execution) thread for each block.

Different always blocks in a module are assumed to have different program counters.

3.1 The simulation cycle

The state of a module during simulation consists of the simulation time (a non-negative integer), a state specifying the values of all variables (inputs, registers and wires) and the value of the program counter of each thread.

It is assumed that the environment to the module (which would normally be other Verilog code – e.g. a test harness) supplies a value for each inputs at each instant of simulation time.

During each simulation cycle a set of pending non-blocking assignments is accumulated. They are executed when there are no more enabled threads.

The instruction pointed to by the program counter of a thread is called the current instruction. A thread is called *waiting* if its current instruction is a timing control instruction `@(`$\mathcal{T}$`)`, otherwise it is called *enabled*.

The concept of an event expression *firing* is defined as follows:

- $\mathcal{V}$ fires if the current value of $\mathcal{V}$ differs from the previous one;
- `posedge` $\mathcal{V}$ fires if the current value of $\mathcal{V}$ is 1 and the previous values was not 1;
- `negedge` $\mathcal{V}$ fires if the current value of $\mathcal{V}$ is 0 and the previous values was not 0;

- $\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$ fires if any of the $\mathcal{T}_i$ fires $(1 \leq i \leq n)$.

Note that an event expression can only fire during or after the second simulation cycle (since there needs to be both a current state and a previous state). A waiting thread fires whenever its event expression fires.

Inititially simulation time is 0, each program counter is set to 0 and each variable (both registers and inputs) has an 'undefined' value `x`. The simulation cycle is as follows:

1 If there are no enabled threads then go to **3**, else non-deterministically choose an enabled thread and execute its current instruction as follows:

(a) $\mathcal{V}$ `=` $\mathcal{E}$ — change the $\mathcal{V}$ component of the state to the value of $\mathcal{E}$ in the current state, increment the program counter and then go to **2**;

(b) $\mathcal{V}$ `<=` $\mathcal{E}$ — add $\mathcal{V}$ `<=` $\mathcal{E}'$ to the list of pending non-blocking assignments, where $\mathcal{E}'$ is the value of $\mathcal{E}$ in the current state (override any previously generated pending addignments to $\mathcal{V}$), increment the program counter and then go to **1**;

(c) `go` n — set the program counter to n then go to **1**;

(d) `ifnot` $\mathcal{E}$ `go` n — if $\mathcal{E}'$ `=== 1`, where $\mathcal{E}'$ is the value of $\mathcal{E}$ in the current state, then increment the program counter, otherwise set it to n, then go to **1**;

2 Increment the program counters of all threads whose current instruction is a timing control that fires and then go to **1**.

3 If there are no pending non-blocking assignments then go to **4**, else execute all pending non-blocking assignments (in any order and overriding any assignments in the state) then go to **2**.

4 Increment the simulation time, update the state with any changes from the inputs and go to **2**.

3.2 Examples

To illustrate the simulation cycle, a number of simple examples will be analysed. The first four will be analysed in more detail than the rest.

A 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 $\mathcal{E}_i$ $(1 \leq i \leq n)$.

3.2.1 Asynchronous timing control (single thread)

A combinational adder (Example 2.2.5 on page 7) is specified by:

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

and translates to

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

If there is only this thread and if `b` and `c` are inputs and `a` a state variable, then the initial state might be:

$\textbf{Time} = 0 \qquad \{\mathtt{pc} \mapsto \mathtt{0}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{0}, \mathtt{c} \mapsto \mathtt{0}\}$

where **Time** indicates the simulation time. At the start of simulation there is an empty set of pending non-blocking assignments. The simulation cycle starts at $\boxed{\mathbf{1}}$. The current instruction is `0`, which is a timing control and so the thread is not enabled. The cycle moves to $\boxed{\mathbf{3}}$ and then, as there are no pending non-blocking assignments, to $\boxed{\mathbf{4}}$.

Suppose `b` changes to `2` at the next simulation time, so the new state is:

$\textbf{Time} = 1 \qquad \{\mathtt{pc} \mapsto \mathtt{0}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{0}\}$

Control passes from $\boxed{\mathbf{4}}$ to $\boxed{\mathbf{2}}$. The timing control `@(b or c)` fires, so the program counter is incremented to get a new state:

$\textbf{Time} = 1 \qquad \{\texttt{pc} \mapsto 1, \texttt{a} \mapsto \texttt{x}, \texttt{b} \mapsto 2, \texttt{c} \mapsto 0\}$

and simulation returns to **1**. Now the current instruction `1` is enabled so, by **1**(a), the value of `a` is updated in the state and the program counter incremented to get:

$\textbf{Time} = 1 \qquad \{\texttt{pc} \mapsto 2, \texttt{a} \mapsto 2, \texttt{b} \mapsto 2, \texttt{c} \mapsto 0\}$

Simulation moves to **2** and then, as nothing fires, to **1**. The current instruction is now the jump `go 0`, which is enabled and so, by **1**(c), the program counter is set to `0` to get a new state:

$\textbf{Time} = 1 \qquad \{\texttt{pc} \mapsto 0, \texttt{a} \mapsto 2, \texttt{b} \mapsto 2, \texttt{c} \mapsto 0\}$

and simulation moves to **2** and then, as there are no other threads to fire, to **1**. The thread is no longer enabled, so simulation moves to **3** and then to **4** and the simulation cycle quiesces and so ends.

Thus, at the end of the cycle the value of `a` is the sum of the values of `b` and `c`. In general, it seems clear that a single thread

`always @(`$\mathcal{T}_{\mathcal{E}}$`)` $\mathcal{W}$ `=` $\mathcal{E}$

will simulate so that the value of $\mathcal{W}$ at the end of the cycle will be the value of $\mathcal{E}$ at its start.

3.2.2 Asynchronous timing control (disjoint threads)

Consider now two threads:

```
forever @(b or c) a1 = b + c;

forever @(b) a2 = b + 1;
```

These translate to:

Thread 1		Thread 2	
0:	`@(b or c)`	0:	`@(b)`
1:	`a1 = b + c`	1:	`a2 = b + 1`
2:	`go 0`	2:	`go 0`

Suppose there are only these two threads, `b` and `c` are inputs, `a1` and `a2` are state variable, `pc1` and `pc2` are the program counters for the two threads listed above, respectively, and the initial state is:

$$\textbf{Time} = 0 \qquad \{\mathtt{pc1} = 0, \mathtt{pc2} = 0, \mathtt{a1} \mapsto \mathtt{X}, \mathtt{a2} \mapsto \mathtt{X}, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0\}$$

Initially nether thread is enabled, so simulation time advances. Suppose `b` changes to 2 as before, so the new state is:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 0, \mathtt{pc2} = 0, \mathtt{a1} \mapsto \mathtt{X}, \mathtt{a2} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 0\}$$

Control passes from **4** to **2**. Both the timing control `@(b or c)` and `@(b)` fire, so both program counters are incremented to get a new state:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 1, \mathtt{pc2} = 1, \mathtt{a1} \mapsto \mathtt{X}, \mathtt{a2} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 0\}$$

and simulation returns to **1**. Now both current instructions are enabled, so a non-deterministic choice is made of which thread to advance. Suppose thread 1 is chosen. The value of `a1` is updated and the program counter incremented to get:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 2, \mathtt{pc2} = 1, \mathtt{a1} \mapsto 2, \mathtt{a2} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 0\}$$

and simulation moves to **2**. The change of `a1` from `X` to 2 doesn't fire any timing controls and so simulation moves to **1**. Thread 1 is still enabled, so a non-derterministic choice must be made. Suppose now thread 2 is chosen: `a2` will be set to the value of `b + 1`, i.e. `3`, and `pc2` incremented. The resulting state is:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 2, \mathtt{pc2} = 2, \mathtt{a1} \mapsto 2, \mathtt{a2} \mapsto 3, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 0\}$$

Simulation moves to **2**, nothing fires, so it moves to **1**. Both threads are still enabled so, in a non-deterministic order, first one program counter and then the other one will be set to `0`. The cycle now quiesces in the state:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 0, \mathtt{pc2} = 0, \mathtt{a1} \mapsto 2, \mathtt{a2} \mapsto 3, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 0\}$$

Thus, at the end of the cycle the value of `a1` is the sum of the values of `b` and `c` and the value of `a2` is one plus the value of `b`. Various non-deterministic choices were made, but it is clear that if different choices were made the resulting state at the end of the cycle would be the same. In general, it seems clear that two *disjoint* threads

`always @(`$\mathcal{T}_{\mathcal{E}_1}$`)` $\mathcal{W}_1$ `=` $\mathcal{E}_1$

`always @(`$\mathcal{T}_{\mathcal{E}_2}$`)` $\mathcal{W}_2$ `=` $\mathcal{E}_2$

will simulate so that for $i = 1, 2$ the value of $\mathcal{W}_i$ at the end of the cycle will, respectively, be the value of $\mathcal{E}_i$ at its start. Disjointness means that the wires being written (viz. $\mathcal{W}_1$ and $\mathcal{W}_2$) do not occur in the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$.

3.2.3 Asynchronous timing control (interacting threads)

Consider now two threads in which `b` is an input and `a` and `c` registers.

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

forever @(b) c = b + 1;
```

Note that first thread reads the register `c` written by the second one. These translate to:

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

Suppose there are only these two threads with program counters `pc1` and `pc2` and the initial state is:

$$\textbf{Time} = 0 \qquad \{\mathtt{pc1} = 0, \mathtt{pc2} = 0, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto \mathtt{X}\}$$

Initially nether thread is enabled, so simulation time advances. Suppose `b` changes to `2` as before, so the new state is:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 0, \mathtt{pc2} = 0, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \mathtt{X}\}$$

Control passes from **4** to **2**. Both the timing control `@(b or c)` and `@(b)` fire, so both program counters are incremented to get a new state:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 1, \mathtt{pc2} = 1, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \mathtt{X}\}$$

and simulation returns to **1**. Now both current instructions are enabled, so a non-deterministic choice is made of which thread to advance. Suppose thread 1 is chosen. The value of `a` is updated (assume `2+X` $=$ `X`) and the program counter incremented to get:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = 2, \mathtt{pc2} = 1, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \mathtt{X}\}$$

and simulation moves to **2** and then **1**. Thread 1 is still enabled, so a non-derterministic choice must be made. Suppose now thread 2 is chosen: `c` will be set to the value of `b + 1`, i.e. `3`, and `pc2` incremented. The resulting state is:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{2}, \mathtt{pc2} = \mathtt{2}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{3}\}$$

Simulation moves to **2**, nothing fires, so it moves to **1**. Both threads are still enabled, so in a non-deterministic order first one program counter and then the other one will be set to `0`. The cycle now quiesces in the state:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{0}, \mathtt{pc2} = \mathtt{0}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{3}\}$$

Thus, at the end of the cycle the value of `a` is undefined and the value of `c` is one plus the value of `b`.

Suppose now that when the state was

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{1}, \mathtt{pc2} = \mathtt{1}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{X}\}$$

thread 2 had been chosen. The value of `c` would be updated and the program counter incremented to get:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{2}, \mathtt{pc2} = \mathtt{1}, \mathtt{a} \mapsto \mathtt{X}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{3}\}$$

and simulation moves to **2** and then **1**. Suppose now thread 1 is chosen: `a` will be set to the value of `b + c`, i.e. `5`, and `pc1` incremented. The resulting state is:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{2}, \mathtt{pc2} = \mathtt{2}, \mathtt{a} \mapsto \mathtt{5}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{3}\}$$

Eventually the cycle will quiesces in a different state:

$$\textbf{Time} = 1 \qquad \{\mathtt{pc1} = \mathtt{0}, \mathtt{pc2} = \mathtt{0}, \mathtt{a} \mapsto \mathtt{5}, \mathtt{b} \mapsto \mathtt{2}, \mathtt{c} \mapsto \mathtt{3}\}$$

Thus in this case the result depends on the non-deterministic choices made.

3.2.4 Asynchronous timing control (latch inference)

The significant feature of the following example is that for some combinations of the inputs (viz. when `clk` is false) the value of the output `q` is not driven. Since `q` is a register this means that it retains its value from the previous simulation cycle, so a hardware synthesiser must generate a latch.

The Verilog source is:

```
forever @(clk or d) if (clk) q = d;
```

which translates to:

```
0:   @(clk or d)
1:   ifnot clk go 3
2:   q = d
3:   go 0
```

Suppose this is the only thread being simulated and that the state at the end of the cycle at simulation time n is:

$$\textbf{Time} = n \qquad \{\texttt{pc} \mapsto \texttt{0}, \texttt{clk} \mapsto \texttt{0}, \texttt{d} \mapsto d, \texttt{q} \mapsto q\}$$

here d and q are the values of input `d` and output `q`, respectively. Their exact values are unimportant.

Suppose now that `clk` goes to `1` at time $n+1$, so at the start of the simulation cycle the state is:

$$\textbf{Time} = n+1 \qquad \{\texttt{pc} \mapsto \texttt{0}, \texttt{clk} \mapsto \texttt{1}, \texttt{d} \mapsto d, \texttt{q} \mapsto q\}$$

The simulation at time $n+1$ will start and **2** and the timing control `@(clk or d)` will fire, so simulation will move to **1** with state:

$$\textbf{Time} = n+1 \qquad \{\texttt{pc} \mapsto \texttt{1}, \texttt{clk} \mapsto \texttt{1}, \texttt{d} \mapsto d, \texttt{q} \mapsto q\}$$

By **1**(d) (as `clk === 1`) the program counter is incremented and simulation returns to **1** with state

$$\textbf{Time} = n+1 \qquad \{\texttt{pc} \mapsto \texttt{2}, \texttt{clk} \mapsto \texttt{1}, \texttt{d} \mapsto d, \texttt{q} \mapsto q\}$$

`q` is then updated to d and (after a few more steps) the cycle quiesces in state

$$\textbf{Time} = n+1 \qquad \{\texttt{pc} \mapsto \texttt{0}, \texttt{clk} \mapsto \texttt{1}, \texttt{d} \mapsto d, \texttt{q} \mapsto d\}$$

If `clk` stays at `1` and `d` changes, then `q` will be updated to `d`'s new value.

If `clk` falls to `0`, then the assignment `q = d` will be jumped over and any changes to `d` ignored.

Thus then `clk` is `1` the output `q` is combinationally driven by `d`, but as soon as `clk` drops to `0` the value of `d` at the last simulation time when `clk` was `1` is latched and drives `q`.

Thus

```
forever @(clk or d) if (clk) q = d;
```

is a transparent latch with clock line `clk` and gate input `d`.

3.2.5 Synchronous timing control (flip-flop)

Consider:

```
forever @(posedge clk) q = d;
```

which translates to:

```
0:   @(posedge clk)
1:   q = d
2:   go 0
```

Whenever the input `clk` changes to `1` the output `q` is set to the value of `q`, and then simulation quiesces.

This is just the behaviour of an edge-triggered flip-flop.

3.2.6 Two flip-flops in series

Consider:

```
forever @(posedge clk) i = d;

forever @(posedge clk) q = i;
```

which translates to two threads

```
0:   @(posedge clk)      0:   @(posedge clk)
1:   i = d               1:   q = i
2:   go 0                2:   go 0
```

Whenever `posedge clk` fires there is a race between `i = d` and `q = i`. If `i = d` is done first then `q` ends up with the value of `d`. If `q = i` is done first then `q` ends up with the previous value of `i`.

Synthesizers usually generate two flip-flops in series, which correspond to `q = i` being done first.

The event semantics can be made unambiguous by changing `i = d` to `i <= d`, so that `q = i` is done before `i` is updated.

The trace semantics given in Chapter 4 currently does not correspond to the synthesis semantics – `q` is assigned `d` (see 4.4.3. This may change.

3.2.7 Synchronous timing control (flip-flop with built-in multiplexer)

Consider:

```
forever @(posedge clk) q <= a ? b : c;
```

which translates to:

```
0:   @(posedge clk)
1:   q = a ? b : c
3:   go 0
```

Whenever the `posedge clk` fires the output `q` is set to the value of `a ? b : c`.

This is just the behaviour of an edge-triggered flip-flop whose input is connected to the output of a combinational multiplexer.

3.2.8 Synchronous timing control (flip-flop with separate multiplexer)

Consider:

```
forever @(posedge clk) q <= d;

forever @(a or b or c) if (a) d = b; else d = c;
```

which translates to two threads:

```
0:   @(posedge clk)
1:   q = d
2:   go 0

0:   @(a or b or c)
1:   ifnot a go 4
2:   d = b
3:   go 5
4:   d = c
5:   go 0
```

In any cycle in which `@(posedge clk)` fires, but `@(a or b or c)` doesn't `q` will be updated with the value of `d`.

In any cycle in which `@(a or b or c)` fires, but `@(posedge clk)` doesn't `d` will be updated with the value of `a ? b : c`.

If both `@(posedge clk)` and `@(a or b or c)` fire at the same time then there is a race condition. If `d` is updated in the second thread before before `q` is updated in the first thread, then the result is to set `q` to the value of `a ? b : c`. However, if the first thread runs faster and `q` is updated before `d` then `q` will end up with the previous value of `d`. In both cases `d` will get the value `a ? b : c`.

If the values on the internal line `d` are ignored, this example behaves like the previous example as long as none of `a`, `b` or `c` change at the same time as a rising edge on `clk`.

Chapter 4

Trace Semantics

The event semantics describes the execution of Verilog in terms of the propagation of changes to wires and registers (i.e. events) via a simulation cycle. Thus event semantics is 'edge-oriented'.

The term *trace semantics* will be used here to mean a semantics that describes the execution of Verilog in terms of sequences of states at successive simulation times.

In general, Verilog programs can have race conditions which make them non-deterministic. The evolution of states is thus a tree rather than a sequence. However, it is desirable that synthesized hardware be deterministic, so race conditions will be excluded by (as yet unformulated) syntactic restrictions. One of our goals, not addressed in this document, is to prove that the restrictions guarantee consistency of the event and trace semantics (and hence guarantee determinacy of the event semantics). This will be a step towards establishing that synthesised hardware will simulate the same as the source (when the restrictions are obeyed).

The clock cycle semantics is obtained from the trace semantics by temporal abstraction [6] on clock edges. The kind of edge to abstract on (i.e. `posedge` or `negedge`) depends on the particular components used. The abstraction to cycle semantics is thus component dependent. With some kinds of components (e.g. transparent level sensitive latches) the abstraction to the clock cycle level is problematical. In contrast, the trace semantics is meaningful for all common components used in clocked synchronous design.

The extraction of the trace semantics is based on the computation of *steps*[1]. A step describes the cumulative effect of a sequence of simulation events that are started by the firing of a timing control and ended when another timing

[1] "Steps" were called "next-state assertions" in an earlier version of this document – further name changes are possible.

control is reached. Steps are obtained by symbolically executing pseudo code starting from a timing control instruction.

The steps provide a compact representation of the event semantics of each always block considered in isolation (i.e. ignoring interleaving within a single simulation cycle). If programs satisfy suitable syntactic restrictions guaranteeing non-interference, then it is hoped to prove that the steps are a correct description of the event semantics.

The trace semantics of a module is represented by a set of sequences of states (a trace being a sequence of states). In general, different traces are obtained with different inputs. The state consists of the registers written by assignments in each always block together with additional control registers, called program counters. Program counters are local to each block. In the initial state program counters are assumed to be `0` and each input and register is assumed to contain `x`.

The traces will be characterised by interpreting the steps as constraints that relate the values of variables to each other, either at the same simulation time or at the preceding time. Steps are written in an explicit-state style of Verilog. For example, the step extracted from:

```
always
 begin
  @(posedge clk) tmp = d1;
  @(negedge clk) r = tmp + d2;
 end
```

is:

```
case (pc)
 0 : @(posedge clk)
        begin
           pc <= 1;
           tmp <= d1;
           r <= previous(r);
        end
 1 : @(negedge clk)
        begin
           pc <= 0;
           tmp <= previous(tmp);
           r <= previous(tmp) + d2;
        end
endcase
```

This step should be read as

$$
\begin{array}{l}
\exists \mathtt{pc}.\\
\;\mathtt{pc(0)=0} \wedge \mathtt{clk(0)=X} \wedge \mathtt{d1(0)=X} \wedge \mathtt{d2(0)=X} \wedge \mathtt{tmp(0)=X} \wedge \mathtt{r(0)=X} \wedge\\
\;\forall t\mathtt{>0}.\\
\quad \mathtt{pc(}t\mathtt{-1)=0} \Rightarrow \mathit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{1} \wedge \mathtt{clk(}t\mathtt{)=1}\\
\qquad\qquad\qquad \mathit{then}\ \mathtt{pc(}t\mathtt{)=1} \wedge \mathtt{tmp(}t\mathtt{)=d1(}t\mathtt{)} \wedge \mathtt{r(}t\mathtt{)=r(}t\mathtt{-1)}\\
\qquad\qquad\qquad \mathit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \mathtt{r(}t\mathtt{)=r(}t\mathtt{-1)}\\
\quad \wedge\\
\quad \mathtt{pc(}t\mathtt{-1)=1} \Rightarrow \mathit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{0} \wedge \mathtt{clk(}t\mathtt{)=0}\\
\qquad\qquad\qquad \mathit{then}\ \mathtt{pc(}t\mathtt{)=0} \wedge \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \mathtt{r(}t\mathtt{)=tmp(}t\mathtt{-1)+d2(}t\mathtt{)}\\
\qquad\qquad\qquad \mathit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \mathtt{tmp(}t\mathtt{)=tmp(}t\mathtt{-1)} \wedge \mathtt{r(}t\mathtt{)=r(}t\mathtt{-1)}
\end{array}
$$

Here the logical variable t ranges over simulation times. The formula above asserts that at any time t greater than 0:

1. if the value of `pc` at t`-1` is `0` then:
   (a) if there is a positive edge on `clk` ending at t then set `pc` to `1`, set `tmp` to the input on `d1` and keep the value of `r` at its previous value;
   (b) if there is no positive edge then `pc`, `tmp` and `r` keep their previous values;
2. if the value of `pc` at t`-1` is `1` then:
   (a) if there is a negative edge on `clk` ending at t then set `pc` to `0`, keep `tmp` at its previous value and set output `r` to the sum of the previous value of `tmp` and the current value of the input `d2`;
   (b) if there is no positive edge then `pc`, `tmp` and `r` keep their previous values.

At time `0` the program counter `pc` is initialised to `0` and all the inputs and registers to `X`.

Given the values of the inputs `clk`, `d1` and `d2` for all times $t > 0$, this formula uniquely determines the values of `pc`, `tmp` and `r` at all times $t > 0$.

Note that the free variables of the formula are `clk`, `d1`, `d2`, `tmp` and `r`. The program counter `pc` is made 'local' by existential quantification. Variables local to a module (i.e. not inputs or outputs) will also be existentially quantified. The time variable t is universally quantified.

A function declaration like

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

generates an equation that of the form

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

where $\mathcal{E}$ is obtained from the function body $\mathcal{S}$. This equation is then used to eliminate (inline) function calls.

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 step:

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

and hence the equation:

$\mathtt{f}(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) \ = \ \mathtt{b\ ?\ c\ ?\ d\ :\ !d\ :\ a}.$

How this equation is derived is explained in a bit more detail later.

4.1 Examples

The examples in this section are intended to convey the idea of steps.

4.1.1

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:

```
case (pc)
 0 : @(posedge clk)
       begin
          pc <= 1;
          a <= 0;
          b <= previous(b);
       end
 1 : @(posedge clk)
       begin
          pc <= 0;
          a <= previous(a);
          b <= previous(a);
       end
endcase
```

4.1.2

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 [8].

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

which generates:

```
case (pc)
 0 : @(posedge clk)
       begin
          pc <= 1;
          total <= data;
       end
 1 : @(posedge clk)
       begin
          pc <= 2;
          total <= previous(total) + data;
       end
 2 : @(posedge clk)
       begin
          pc <= 0;
          total <= previous(total) + data;
       end
endcase
```

4.1.3

An explicit style of description of the machine in Example 4.1.2 is given next. This is Example 8-17 from the Synopsys HDL Compiler for Verilog Reference Manual [8].

```
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:

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

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.

4.1.4

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

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

4.1.5

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

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

generates

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

Since whenever `b` and `c` change, `a` is updated, it follows (hopefully by induction over time – details to be worked out elsewhere) that this step is equivalent to the equation `a = b+c`. However consider instead:

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

which generates:

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

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.

4.1.6

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:

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

4.1.7

The sequential block in Example4.1.6, 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 28. This statement is not an always block, so it does not translate to an infinite loop. Its translation to pseudo-code is:

```
0:   f = a
1:   ifnot b go 6
2:   ifnot c go 5
3:   f = d
4:   go 6
5:   f = !d
```

It is equivalent to a single assignment done once. On the next cycle the program counter, instead of pointing at the beginning of the program again, points outside the pseudo-code (to instruction `6`, which is renumbered to instruction `1` after symbolic execution).

The pseudo-code symbolically executes to:

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

The expression assigned to the function name `f` is used to generate the equation defining `f` (see page 39 at the end of 4.2).

If the assignment `f = a;` is deleted the resulting step becomes:

```
case (pc)
 0 : begin
        pc <= 1;
        f <= b ? c ? d : !d : X;
     end
endcase
```

This shows that for some combinations of inputs the function is 'not defined' – i.e. returns `X`.

4.1.8

Each step, 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.

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

generates:

```
case (pc)
 0 : begin
        pc <= p ? 1 : 3;
        c <= X;
        a <= p ? 1 : 5;
        b <= X;
     end
 1 : @(posedge clk)
        begin
           pc <= 2;
           c <= previous(c);
           a <= previous(a);
           b <= 2;
        end
 2 : @(negedge clk)
        begin
           pc <= p ? 1 : 3;
           c <= 3;
           a <= p ? 1 : 5;
           b <= previous(b);
        end
 3 : @(clk)
        begin
           pc <= p ? 1 : 3;
           c <= previous(c);
           a <= p ? 1 : 5;
           b <= 6;
        end
endcase
```

4.2 Symbolic execution

Steps 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, `pc`, called the program counter is assumed.

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 `V0`.

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 variables $\mathcal{V}_1, \ldots, \mathcal{V}_n$ in an expression $\mathcal{E}$ is denoted by:

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

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 $\mathtt{pc} \mapsto i$ 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 $\{\mathtt{pc} \mapsto j,\ \mathcal{R}_1 \mapsto \mathcal{E}_1^f,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}_n^f\}$ be the state resulting from recursively symbolically executing with $\mathtt{pc} \mapsto n$;
   - let $\{\mathtt{pc} \mapsto k,\ \mathcal{R}_1 \mapsto \mathcal{E}_1^t,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}_n^t\}$ be the state resulting from recursively symbolically executing with $\mathtt{pc} \mapsto i+1$;
   - return as the result of the symbolic execution the state $\{\mathtt{pc} \mapsto \mathcal{E}'\ ?\ k\ :\ j,\ \mathcal{R}_1 \mapsto \mathcal{E}'\ ?\ \mathcal{E}'^t_1\ :\ \mathcal{E}'^f_1,\ \ldots\ ,\ \mathcal{R}_n \mapsto \mathcal{E}'\ ?\ \mathcal{E}'^t_n\ :\ \mathcal{E}'^f_n\}$

6. The instruction `disable` $\mathcal{B}$ should not be generated. `V0` 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 a step from a statement as follows.

If the first instruction is not a timing control, then generate a case item:

$\mathtt{0\ :\ begin\ pc\ <=}\ j;\ \mathcal{R}_1\ \mathtt{<=}\ \mathcal{E}_1;\ \ldots\ ;\ \mathcal{R}_n\ \mathtt{<=}\ \mathcal{E}_n;\ \mathtt{end}$

where $\{\mathtt{pc} \mapsto j, \mathcal{R}_1 \mapsto \mathcal{E}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{E}_n\}$ is the state resulting from symbolic execution starting with $\{\mathtt{pc} \mapsto \mathtt{0}, \mathcal{R}_1 \mapsto \mathtt{X}, \ldots, \mathcal{R}_n \mapsto \mathtt{X}\}$ 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 $\mathtt{@}(\mathcal{T})$ generate a case item:

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

where $\{\mathtt{pc} \mapsto j, \mathcal{R}_1 \mapsto \mathcal{E}_1, \ldots, \mathcal{R}_n \mapsto \mathcal{E}_n\}$ results from symbolic execution starting with $\{\mathtt{pc} \mapsto i+1, \mathcal{R}_1 \mapsto \mathtt{previous}(\mathcal{R}_1), \ldots, \mathcal{R}_n \mapsto \mathtt{previous}(\mathcal{R}_\mathtt{n})\}$ and the empty set of pending non-blocking assignments.

The step from an always block `always` $\mathcal{S}$ is obtained by generating the step from the statement `forever` $\mathcal{S}$. In the examples shown earlier in 4.1, the values that the program counter ranges over have been compacted to a contiguous sequence of numbers starting from `0`.

4.3 The meaning of a module

A module in V0 has the general form:

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

In V0, there is no semantic difference between wires and registers and a continuous assignment `assign` $\mathcal{W}$ `=` $\mathcal{E}$ is considered to be the always block `always @(`$\mathcal{T}_{\mathcal{E}}$`)` $\mathcal{W}$ `=` $\mathcal{E}$ where $\mathcal{T}_{\mathcal{E}}$ is $\mathcal{V}_1$ `or` $\cdots$ `or` $\mathcal{V}_n$ ($\mathcal{V}_1$, …, $\mathcal{V}_n$ being the variables occurring in $\mathcal{E}$).

Function calls are eliminated by replacing (inlining) them with expressions obtained from the step extracted from the function body. The equation for:

$$
\begin{array}{l}
\texttt{function}\ \mathcal{F}\texttt{;} \\
\quad \texttt{input}\ \mathcal{V}_1\texttt{;}\ \ldots\ \mathcal{V}_n\texttt{;} \\
\quad \mathcal{S} \\
\texttt{endfunction}
\end{array}
$$

is obtained by generating the step from the body $\mathcal{S}$ which, if the function is well-formed, should be of the form:

```
case (pc)
 0 :  begin
        pc <= 1;
           ⋮
        F <= E;
           ⋮
      end
endcase
```

The equation defining $\mathcal{F}$ is then:

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

Instances of the left hand side – i.e. function calls – can be eliminated by replacing them with the corresponding instance of the right hand side.

The trace semantics of a module is defined by the conjuntion of the predicates corresponding to the steps, after continuous assignments and function calls have been eliminated.

4.4 Examples of trace semantics

This section contains some examples to illustrate trace semantics. The semantics will first be expressed directly in terms of predicates on traces using explicit quantification over time and then in a more compact form (with no explicit time) using abbreviations based on temporal logic. The direct form can be generated uniformly from Verilog via the computation of steps. It is not clear whether the temporal logic form without explicit time variables can be uniformly generated for arbitrary Verilog, but it is plausible that it can be generated for the synthesizable subset.

The temporal abbreviations use constants and logical operators 'lifted' pointwise to predicates. These are denoted with bigger and bolder versions of the normal symbols, for example:

- $\forall\, t.\ \mathbf{1}(t) \;=\; \mathtt{1}$
- $\forall\, t.\ (f_1 \mathbf{+} f_2)(t) \;=\; f_1(t){+}f_2(t)$
- $\forall\, t.\ (f_1 \mathbf{?} f_2 \mathbf{:} f_3)(t) \;=\; f_1(t)?f_2(t){:}f_3(t)$

Two traces are equal if and only if they are equal as functions, i.e. equal at all times:

- $f_1 \equiv f_2$ means $\forall\, t.\ f_1(t) = f_2(t)$

Latches and flip-flops 'freeze' the values of variables to the value they had the last time an event (edge) happened.

Suppose p represents a set of events in the sense that $p\ t$ is true exactly when an event happens at time t. Define `last` $p\ f\ t$ to be the value of f at the last time before (or including) t that p was true. If p is not true at any time up to and including t, then `last` $p\ f\ t = \texttt{x}$.

Note that `last` $p\ f$ is a trace that only changes when events specified by p occur and hence `last` p `(last` $p\ f$`)` $\equiv$ `last` $p\ f$.

If f is a trace, then define the trace `previous(`f`)` by:

- `previous(`f`)(0) = x`
- `previous(`f`)(`t`+1) =` f`(`t`)`

If f is a trace, then define the boolean-valued traces `posedge` f and `negedge` f to satisfy:

- `posedge` f `0` is false and
  $\forall\, t > 0.$ `posedge` $f\ t\ =\ (f(t{-}1) \neq 1 \wedge f(t) = 1)$
- `negedge` f `0` is false and
  $\forall\, t > 0.$ `negedge` $f\ t\ =\ (f(t{-}1) \neq 0 \wedge f(t) = 0)$

Thus `last (posedge clk)` $f\ t$ is the value of f at the last time before or equal to t when `clk` has just finished a rising edge – i.e. f`(`t'`)`, where t' is the greatest time $t' \leq t$ such that `clk(`t'`-1)`$\neq$`1` and `clk(`t'`)=1`.

Also observe that `last (posedge clk) (previous(`f`))` t is the value of f at the last time before or equal to t when `clk` has just started a rising edge – i.e. f`(`t'`-1)` where t' is as above.

Warning: the logical manipulations that are asserted to hold for the examples below have not been fully verified. However, it is expected to be straightforward to mechanically check them with a theorem prover. For VFE it is planned to implement a semantics extractor, that automatically derives from a module text a simplified formula representing its trace semantics.

4.4.1 Combinational logic

Here is a combinational incrementer:

```
forever @(i) q = i+1;
```

This generates the step:

```
case (pc)
 0 : @(i)
        begin
           pc <= 0;
           q <= i+1;
        end
endcase
```

which denotes the following trace specification:

$$
\begin{array}{l}
\exists \mathtt{pc}. \\
\ \mathtt{pc(0)=0} \wedge \mathtt{i(0)=X} \wedge \mathtt{q(0)=X} \wedge \\
\ \forall t\mathtt{>0}. \\
\quad \mathtt{pc(}t\mathtt{-1)=0} \Rightarrow \mathit{if}\ \mathtt{i(}t\mathtt{-1)}\neq\mathtt{i(}t\mathtt{)} \\
\qquad\qquad\qquad\quad \mathit{then}\ \mathtt{pc(}t\mathtt{)=0} \wedge \mathtt{q(}t\mathtt{)=i(}t\mathtt{)+1} \\
\qquad\qquad\qquad\quad \mathit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \mathtt{q(}t\mathtt{)=q(}t\mathtt{-1)}
\end{array}
$$

By induction over t this is logically equivalent to:

$$\mathtt{i(0)=X} \wedge \mathtt{q(0)=X} \wedge \forall t\mathtt{>0.}\ \mathtt{q(}t\mathtt{)=i(}t\mathtt{)+1}$$

i.e.:

$$\mathtt{i(0)=X} \wedge \mathtt{q(0)=X} \wedge \mathtt{q} \equiv \mathtt{i} + 1$$

4.4.2 Flip-flops

```
forever @(posedge clk) q = d;
```

generates the step:

```
case (pc)
 0 : @(posedge clk)
        begin
           pc <= 0;
           q <= d;
        end
endcase
```

which denotes the following trace specifications:

$$
\begin{array}{l}
\exists \mathtt{pc}.\\
\quad \mathtt{pc(0)=0} \wedge \mathtt{d(0)=X} \wedge \mathtt{q(0)=X} \wedge\\
\quad \forall t\mathtt{>0}.\\
\quad\quad \mathtt{pc(}t\mathtt{-1)=0} \Rightarrow \mathit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{1} \wedge \mathtt{clk(}t\mathtt{)=1}\\
\quad\quad\quad\quad\quad\quad \mathit{then}\ \mathtt{pc(}t\mathtt{)=0} \wedge \mathtt{q(}t\mathtt{)=d(}t\mathtt{)}\\
\quad\quad\quad\quad\quad\quad \mathit{else}\ \mathtt{pc(}t\mathtt{)=pc(}t\mathtt{-1)} \wedge \mathtt{q(}t\mathtt{)=q(}t\mathtt{-1)}
\end{array}
$$

which is equivalent to:

$$
\begin{array}{l}
\mathtt{d(0)=X} \wedge \mathtt{clk(0)=X} \wedge \mathtt{q(0)=X} \wedge\\
\forall t\mathtt{>0}.\ \mathit{if}\ \mathtt{clk(}t\mathtt{-1)}\neq\mathtt{1} \wedge \mathtt{clk(}t\mathtt{)=1}\\
\quad\quad \mathit{then}\ \mathtt{q(}t\mathtt{)=d(}t\mathtt{)}\\
\quad\quad \mathit{else}\ \mathtt{q(}t\mathtt{)=q(}t\mathtt{-1)}
\end{array}
$$

which is equivalent to:

$$\mathtt{d(0)=X} \wedge \mathtt{clk(0)=X} \wedge \mathtt{q(0)=X} \wedge \mathtt{q} \equiv \mathtt{last(posedge\ clk)d}$$

4.4.3 Flip-flops in series

```
forever @(posedge clk) i = d;

forever @(posedge clk) q = i;
```

generates the steps:

```
case (pc)
 0 : @(posedge clk)
       begin
          pc <= 0;
          i <= d;
       end
endcase

case (pc)
 0 : @(posedge clk)
       begin
          pc <= 0;
          q <= i;
       end
endcase
```

which denote the following trace specifications:

$$
\begin{array}{l}
\exists \texttt{pc}.\\
\quad \texttt{pc(0)=0} \wedge \texttt{d(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\quad \forall t\texttt{>0}.\\
\qquad \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}\\
\qquad\qquad\qquad\qquad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{i(}t\texttt{)=d(}t\texttt{)}\\
\qquad\qquad\qquad\qquad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{i(}t\texttt{)=i(}t\texttt{-1)}
\end{array}
$$

and

$$
\begin{array}{l}
\exists \texttt{pc}.\\
\quad \texttt{pc(0)=0} \wedge \texttt{clk(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\quad \forall t\texttt{>0}.\\
\qquad \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}\\
\qquad\qquad\qquad\qquad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{q(}t\texttt{)=i(}t\texttt{)}\\
\qquad\qquad\qquad\qquad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

which are equivalent to:

$$
\begin{array}{l}
\texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\forall t\texttt{>0}.\ \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}\\
\qquad\quad \mathit{then}\ \texttt{i(}t\texttt{)=d(}t\texttt{)}\\
\qquad\quad \mathit{else}\ \texttt{i(}t\texttt{)=i(}t\texttt{-1)}
\end{array}
$$

and

$\texttt{clk(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge$
$\forall t\texttt{>0}.\ \textit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}$
$\quad\quad \textit{then}\ \texttt{q(}t\texttt{)=i(}t\texttt{)}$
$\quad\quad \textit{else}\ \texttt{q(}t\texttt{)=q(}t\texttt{-1)}$

which, when conjoined together, are equivalent to:

$\texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge$
$\forall t\texttt{>0}.\ \textit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}$
$\quad\quad \textit{then}\ \texttt{q(}t\texttt{)=d(}t\texttt{)} \wedge \texttt{i(}t\texttt{)=d(}t\texttt{)}$
$\quad\quad \textit{else}\ \texttt{q(}t\texttt{)=q(}t\texttt{-1)} \wedge \texttt{i(}t\texttt{)=i(}t\texttt{-1)}$

Using temporal operators, the two components are equivalent to:

$\texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{i(0)=X} \wedge \texttt{i} \equiv \texttt{last(posedge clk)d}$

$\texttt{i(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{q(0)=X} \wedge \texttt{q} \equiv \texttt{last(posedge clk)i}$

Since

```
last(posedge clk)(last(posedge clk)d) = last(posedge clk)d
```

it follows by substitution with respect to $\equiv$ that the conjunction of these is equivalent to:

$\texttt{i(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{q(0)=X} \wedge$
$\texttt{i} \equiv \texttt{last(posedge clk)d} \wedge \texttt{q} \equiv \texttt{last(posedge clk)d}$

4.4.4 Flip-flop with built-in incrementer

```
forever @(posedge clk) q = d+1;
```

generates the step:

```
case ( pc )
 0 : @(posedge clk)
        begin
           pc <= 0;
           q <= d + 1;
        end
endcase
```

which denotes the following trace specification:

$$
\begin{array}{l}
\exists \texttt{pc}. \\
\quad \texttt{pc(0)=0} \wedge \texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\quad \forall t\texttt{>0}. \\
\quad\quad \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1} \\
\qquad\qquad\qquad\qquad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{q(}t\texttt{)=d(}t\texttt{)+1} \\
\qquad\qquad\qquad\qquad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

which simplifies to:

$$
\begin{array}{l}
\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \\
\forall t\texttt{>0}.\ \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1} \\
\qquad\quad \mathit{then}\ \texttt{q(}t\texttt{)=d(}t\texttt{)+1} \\
\qquad\quad \mathit{else}\ \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

Using temporal operators:

$$\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \texttt{q} \equiv \texttt{last(posedge clk)(d} + \mathsf{1}\texttt{)}$$

Note that since $\mathsf{1}$ is a constant trace, the equation with $\equiv$ is equivalent to:

$$\texttt{q} \equiv \texttt{(last(posedge clk))d} + \mathsf{1}$$

4.4.5 Flip-flop with separate incrementer

```
forever @(posedge clk) i = d;

forever @(i) q = i+1;
```

generates the steps:

```
case (pc)
 0 : @(posedge clk)
        begin
           pc <= 0;
           i <= d;
        end
endcase
```

and

```
case (pc)
 0 : @(i)
       begin
          pc <= 0;
          q <= i+1;
       end
endcase
```

which denote the following trace specifications:

$$
\begin{array}{l}
\exists \texttt{pc}.\\
\quad \texttt{pc(0)=0} \wedge \texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{i(0)=X} \wedge\\
\quad \forall t\texttt{>0}.\\
\qquad \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}\\
\qquad\qquad\qquad\qquad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{i(}t\texttt{)=d(}t\texttt{)}\\
\qquad\qquad\qquad\qquad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{i(}t\texttt{)=i(}t\texttt{-1)}
\end{array}
$$

$$
\begin{array}{l}
\exists \texttt{pc}.\\
\quad \texttt{pc(0)=0} \wedge \texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\quad \forall t\texttt{>0}.\\
\qquad \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{i(}t\texttt{-1)}\neq\texttt{i(}t\texttt{)}\\
\qquad\qquad\qquad\qquad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{q(}t\texttt{)=i(}t\texttt{)+1}\\
\qquad\qquad\qquad\qquad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

which simplify to:

$$
\begin{array}{l}
\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{i(0)=X} \wedge\\
\forall t\texttt{>0}.\ \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{1} \wedge \texttt{clk(}t\texttt{)=1}\\
\qquad\quad \mathit{then}\ \texttt{i(}t\texttt{)=d(}t\texttt{)}\\
\qquad\quad \mathit{else}\ \texttt{i(}t\texttt{)=i(}t\texttt{-1)}
\end{array}
$$

$$
\begin{array}{l}
\texttt{i(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\forall t\texttt{>0}.\ \texttt{q(}t\texttt{)=i(}t\texttt{)+1}
\end{array}
$$

If these are conjoined together, and `i` is made local (i.e. existentially quantified) then the result simplifies to:

```
∃i.
 clk(0)=x ∧ d(0)=x ∧ i(0)=x ∧ q(0)=x ∧
 ∀t>0. if clk(t-1)≠1 ∧ clk(t)=1
         then i(t)=d(t)
         else i(t)=i(t-1)
        ∧
        q(t)=i(t)+1
```

which further simplifies to:

```
 clk(0)=x ∧ d(0)=x ∧ q(0)=x ∧
 ∀t>0. if clk(t-1)≠1 ∧ clk(t)=1
         then q(t)=d(t)+1
         else q(t)=q(t-1)
```

which is the same as the flip-flop with a built-in incrementer (previous example).

Using temporal operators, the two components are:

```
 clk(0)=x ∧ d(0)=x ∧ i(0)=x ∧ i ≡ last(posedge clk)d

 i(0)=x ∧ q(0)=x ∧ q ≡ i + 1
```

Conjoining these and existentially quantifying `i` yields:

```
∃i.
 clk(0)=x ∧ d(0)=x ∧ i(0)=x ∧ i ≡ last(posedge clk)d
 ∧
 i(0)=x ∧ q(0)=x ∧ q ≡ i + 1
```

which simplifies to:

```
 clk(0)=x ∧ d(0)=x ∧ q(0)=x ∧ q ≡ last(posedge clk)(d + 1)
```

4.4.6 A Simple Moore machine

The program below describes a machine with a synchronous set. Asserting input `set` sets the state to `0` on the next positive edge of a clock. If `set` is not asserted, then at each positive edge of the clock the value of `q` is set to its current value plus the value being input on `d`.

```
forever
 @(posedge clk)
 if (set) q = 0; else q = q + d;
```

This generates the step:

```
case (pc)
 0 : @(posedge clk)
       begin
          pc <= 0;
          q <= set ? 0 : previous(q) + d;
       end
endcase
```

which denotes the following trace specifications:

```
∃pc.
 pc(0)=0 ∧ d(0)=X ∧ set(0)=X ∧ clk(0)=X ∧ q(0)=X ∧
 ∀t>0.
  pc(t-1)=0 ⇒ if clk(t-1)≠1 ∧ clk(t)=1
               then pc(t)=0 ∧ q(t)=set(t)?0:(q(t-1)+d(t))
               else pc(t)=pc(t-1) ∧ q(t)=q(t-1)
```

which is equivalent to:

```
d(0)=X ∧ set(0)=X ∧ clk(0)=X ∧ q(0)=X ∧
∀t>0. if clk(t-1)≠1 ∧ clk(t)=1
       then q(t)=set(t)?0:(q(t-1)+d(t))
       else q(t)=q(t-1)
```

which is equivalent to:

```
d(0)=X ∧ set(0)=X ∧ clk(0)=X ∧ q(0)=X ∧
q ≡ last(posedge clk)(set ? 0 : (previous(q) + d))
```

4.4.7 Behavioral description of a transparent latch

```
forever @(clk or d) if (clk) q = d;
```

generates the step:

```
case (pc)
 0 : @(clk or d)
        begin
           pc <= 0;
           q <= clk ? d : previous(q);
        end
endcase
```

which denotes:

$$
\begin{array}{l}
\exists \texttt{pc}.\\
\quad \texttt{pc(0)=0} \wedge \texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\quad \forall t\texttt{>0}.\\
\quad\; \texttt{pc(}t\texttt{-1)=0} \Rightarrow \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{clk(}t\texttt{)} \vee \texttt{d(}t\texttt{-1)}\neq\texttt{d(}t\texttt{)}\\
\qquad\qquad\qquad\quad \mathit{then}\ \texttt{pc(}t\texttt{)=0} \wedge \texttt{q(}t\texttt{)=clk(}t\texttt{)?d(}t\texttt{):q(}t\texttt{-1)}\\
\qquad\qquad\qquad\quad \mathit{else}\ \texttt{pc(}t\texttt{)=pc(}t\texttt{-1)} \wedge \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

and simplifies to:

$$
\begin{array}{l}
\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge\\
\forall t\texttt{>0}.\ \mathit{if}\ \texttt{clk(}t\texttt{-1)}\neq\texttt{clk(}t\texttt{)} \vee \texttt{d(}t\texttt{-1)}\neq\texttt{d(}t\texttt{)}\\
\qquad\quad \mathit{then}\ \texttt{q(}t\texttt{)=clk(}t\texttt{)?d(}t\texttt{):q(}t\texttt{-1)}\\
\qquad\quad \mathit{else}\ \texttt{q(}t\texttt{)=q(}t\texttt{-1)}
\end{array}
$$

This is equivalent to:

$$\texttt{clk(0)=X} \wedge \texttt{d(0)=X} \wedge \texttt{q(0)=X} \wedge \texttt{q} \equiv \texttt{clk?d:last(negedge clk)d}$$

4.4.8 Implementation of a transparent latch

A transparent latch can be implemented with a negative edge-triggered flip-flop and some combinational logic to connect the input to the output when the enable signal (`clk`) is high. Assume `i` is local.

```
forever @(negedge clk) i = d;

forever @(clk or d or i) q = clk ? d : i;
```

generates the steps:

```
case (pc)
 0 : @(negedge clk)
        begin
           pc <= 0;
           i <= d;
        end
endcase

case (pc)
 0 : @(clk or d or i)
        begin
           pc <= 0;
           q <= clk ? d : i;
        end
endcase
```

which, taken together, denote:

```
∃i.
∃pc.
 pc(0)=0 ∧ clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧
 ∀t>0.
  pc(t-1)=0 ⇒ if clk(t-1)≠0 ∧ clk(t)=0
              then pc(t)=0 ∧ i(t)=d(t)
              else pc(t)=pc(t-1) ∧ i(t)=i(t-1)
∧
∃pc.
 pc(0)=0 ∧ clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X ∧
 ∀t>0.
  pc(t-1)=0 ⇒ if clk(t-1)≠clk(t) ∨ d(t-1)≠d(t) ∨ i(t-1)≠i(t)
              then pc(t)=0 ∧ q(t)=clk(t)?d(t):i(t)
              else pc(t)=pc(t-1) ∧ q(t)=q(t-1)
```

which simplifies to:

```
∃i.
 clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X
 ∧
 ∀t>0. if clk(t-1)≠0 ∧ clk(t)=0
        then i(t)=d(t)
        else i(t)=i(t-1)
       ∧
       if clk(t-1)≠clk(t) ∨ d(t-1)≠d(t) ∨ i(t-1)≠i(t)
        then q(t)=clk(t)?d(t):i(t)
        else q(t)=q(t-1)
```

One would hope to be able to show this equivalent to the trace semantics of the behavioural description of a transparent latch (previous example), but the equivalence is not immediately obvious. However, working with temporal operators makes the equivalence clearer.

The two components are:

```
clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ i ≡ last(negedge clk)d

clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X ∧ q ≡ clk ? d : i
```

Conjoining these and existentially quantifying `i` yields:

```
∃i.
 clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ i ≡ last(negedge clk)d
 ∧
 clk(0)=X ∧ d(0)=X ∧ i(0)=X ∧ q(0)=X ∧ q ≡ clk ? d : i
```

which simplifies, via substitution with respect to $\equiv$, to:

```
clk(0)=X ∧ d(0)=X ∧ q(0)=X ∧ q ≡ clk ? d : last(negedge clk)d
```

Chapter 5

Cycle Semantics

The cycle semantics is a sequential machine (a Mealy machine, in general) whose state transitions are determined by a clock. Only certain programs can be sensibly interpreted as clocked sequential machines, and so only a subset of `V0` has a cycle semantics. Furthermore, the kind of clock events that clock the system depends on the design style. For `V0`, it is assumed that clocking is done on the positive edge of a single global clock called `clk`.

A module has a cycle semantics if its trace semantics can be expressed as a conjunction of combinational equations and next-state assertions.

Combinational equations have the form

$$V \equiv M$$

where V is a variable and M is an expression built out of trace constants (like `0`, `1` etc), input and state variables using trace operators (like $+$, $-\,?\,-\,:\,-$ etc).

Next state assertions have the form:

$$V \equiv \texttt{last(posedge clk)}\,N$$

where V is a state variable and N is an expression built out of trace constants (like `0`, `1` etc), input variables and expressions of the form `previous(`S`)` (where S is a state variable) using trace operators (like $+$, $-\,?\,-\,:\,-$ etc).

If the input variables are $I_1, \ldots, I_m$, the state variables are $S_1, \ldots, S_n$ and the output variables are $O_1, \ldots, O_p$ then the general form required for cycle semantics extraction is:

$$
\begin{array}{l}
O_1 \equiv M_1[I_1,\ldots,I_m,S_1,\ldots,S_n] \\
\quad\wedge \\
\quad\vdots \\
\quad\wedge \\
O_p \equiv M_p[I_1,\ldots,I_m,S_1,\ldots,S_n] \\
\quad\wedge \\
S_1 \ \equiv \texttt{last(posedge clk)}\,N_1[I_1,\ldots,I_m,\texttt{previous(}S_1\texttt{)},\ldots,\texttt{previous(}S_n\texttt{)}] \\
\quad\wedge \\
\quad\vdots \\
\quad\wedge \\
S_n \ \equiv \texttt{last(posedge clk)}\,N_n[I_1,\ldots,I_m,\texttt{previous(}S_1\texttt{)},\ldots,\texttt{previous(}S_n\texttt{)}]
\end{array}
$$

Here $M_i[I_1,\ldots,I_m,S_1,\ldots,S_n]$ indicates an expression built using $\mathsf{0}$, $\mathsf{1}$, $+$, $-\,\mathsf{?}-\mathsf{:}-$ etc and the variables indicated between the square brackets. Similarly for N_i.

The machine specified by a conjunction in the form above has output function:

$$\langle i_1,\ldots,i_m,s_1,\ldots,s_n\rangle \mapsto \langle f_{M_1}(i_1,\ldots,i_m,s_1,\ldots,s_n),\ldots,f_{M_p}(i_1,\ldots,i_m,s_1,\ldots,s_n)\rangle$$

and next-state function:

$$\langle i_1,\ldots,i_m,s_1,\ldots,s_n\rangle \mapsto \langle f_{N_1}(i_1,\ldots,i_m,s_1,\ldots,s_n),\ldots,f_{N_n}(i_1,\ldots,i_m,s_1,\ldots,s_n)\rangle$$

where the functions $f_{M_i},\ldots,f_{M_p},f_{N_1},\ldots,f_{N_n}$ are the 'obvious' ones derived from $M_i,\ldots,M_p,N_1,\ldots,N_n$ by replacing operators like $\mathsf{0}$, $\mathsf{1}$, $+$, $-\,\mathsf{?}-\mathsf{:}-$ by `0`, `1`, `+`, $-$`?`$-$`:`$-$ respectively.

It may be that state variables are also output variables (see the example below), in which case there is no output function.

In the VFE project it is hoped to provide automated tools for manipulating trace semantics into the above form and then to extract the cycle semantics.

It is assumed that initially all variables have the value `x`.

Example

Recall the simple Moore machine:

```
forever
 @(posedge clk)
 if (set) q = 0; else q = q + d;
```

The trace semantics of this (after simplification) was:

$$\texttt{d(0)=X} \wedge \texttt{set(0)=X} \wedge \texttt{clk(0)=X} \wedge \texttt{q(0)=X} \wedge$$
$$\texttt{q} \equiv \texttt{last(posedge clk)(set} \,?\, 0 : \texttt{(previous(q)} + \texttt{d))}$$

It is assumed that the state `q` is output, so there is no output function. The next-state function is given by:

$$f(\texttt{d}, \texttt{set}, \texttt{q}) \;=\; \texttt{set ? 0 : q+d}$$

Bibliography

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

[2] Ching-Tsun Chou. Synchronous verilog: A proposal. Technical Memorandum FLA-ACR-97-01, Fujitsu Laboratories of America, 3350 Scott Blvd., Bldg. 34, Santa Clara, CA 95054, U.S.A., 1997. Available from ftp://ftp.cs.ucla.edu/pub/chou/sv0.ps.gz.

[3] 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.

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

[5] IEEE. P1364 IEEE Draft Verilog Hardware Description Language Reference Manual (LRM). IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[6] Thomas F. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993. ISBN 0-521-41718-X.

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

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

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