Relating event and trace semantics of Hardware Description Languages

Mike Gordon

University of Cambridge Computer Laboratory
Email: Mike.Gordon@cl.cam.ac.uk

Industry standard hardware description languages like Verilog and VHDL are based on event simulation. Formal verification methods such as theorem proving and model checking are based on the analysis of sets of traces of transitions systems or automata. There is a semantic gap between these two interpretations and consequently one cannot be sure that results obtained by simulation and formal verification are consistent. This paper is a step towards bridging this gap. A simple simulation cycle is specified for a Verilog-like language and it is shown how for simple examples of practical interest this can be related to a standard formal verification oriented model based on traces.

Keywords: Hardware description languages, semantics, simulation, formal verification

Received 2 June 2000; accepted 23 August 2000

1. DEDICATION

This paper is dedicated to Graham Birtwistle on the occasion of his 60th birthday. Professor Birtwistle is now at the peak of a distinguished career than has spanned many areas of computer science and engineering. His early work contributed to the development of the Simula language and he made important contributions to the evolution of event simulation. His later work has been on formal verification of processors, both synchronous and asynchronous. The first formal verification of a fabricated processor was performed by Birtwistle's team at the University of Calgary.

This paper aims to connect the worlds of the 'earlier' and 'later' Birtwistle and is a tribute to his exceptional contributions to both worlds.

2. INTRODUCTION

Hardware description languages (HDLs) like VHDL and Verilog have a simulation oriented semantics based on events, i.e. changes to the values of wires and registers. This *event semantics* can accurately model detailed asynchronous behavour, but is very fine-grained and does 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 sequences 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 is an abstraction called the *trace semantics*. If there are race conditions, then there may be several possible successor states to a given state (i.e. branching time). Trace semantics has the same timescale as event (simulation) semantics – namely simulation time – but abstracts away from the individual events within a single simulation cycle (delta-time).

This paper is only about trace semantics, but there are other more abstract models of behaviour. For example, clocked sequential systems can viewed in terms of the sequence of states held in registers during successive clock cycles. This view is called the *cycle semantics*. 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. A cycle semantics determines the exact clock cycle that each register transfer occurs on. A yet more abstract view is *behavioural semantics* (9) in which some sequences of state transitions are merged into 'super transitions' resulting in a temporal abstraction (10) of the hardware behaviour. Unlike a cycle semantics, a behavioural semantics need not specify the clock cycle that an operation occurs on, only that it is performed sometime within a super transition.

The rest of this paper proceeds as follows. First there is a brief review of related work. Next a simple Verilog-like hardware description language (HDL) is described informally and then formally explicated via a translation to a 'semantic pseudo-code'. The intention is to show that a fairly standard semantics-preserving compilation process can reduce HDL programming constructs to a simple canonical form. The compiled form is simpler to study and the compilation shows that studying it does not lose generality. The semantic pseudo code is then given both a formal event semantics via a formal simulation cycle and from this a trace semantics is derived. It is then shown that for a class of programs, together with some reasonable assumptions, the trace semantics derived from the simulation cycle corresponds to the normal formal verification semantics.

3. RELATED WORK

The semantics of Verilog has been specified from a number of perspectives. The official semantics is provided by the IEEE standard (8), but this is imprecise and is not a complete and unambiguous specification. In practice, experiments with particular 'golden' simulators are used to clarify details. There has been relatively little academic semantic research into Verilog. Gordon Pace at Oxford wrote a PhD (11) on modelling the semantics in an extension of the Duration Calculus and he was able to relate denotational and operational semantics of a very small subset and to derive algebraic laws. John Fiskio-Lasseter produced an axiomatic specification of the Verilog simulation cycle as part of his masters thesis at the University of Oregon (4). He was able to formally specify a substantial subset of the language in a way that appears "to match the behaviour of the leading simulation packages". Pace's work is a relatively deep analysis of a small subset whereas Fiskio-Lasseter's is a shallower treatment of a much larger subset. The two studies use completely different semantic frameworks. A third investigation by Bowen, Jifeng and Qiwen (1) used structured operational semantics to model the simulation cycle. They have animated their model using Prolog and performed experiments to investigate the accuracy of the semantics. This work has inspired another duration calculus based denotational semantics that goes beyond Pace's work (7).

Many formal verification tools (6) can accept designs specified in Verilog (and other industry standard HDLs), but the treatment of the semantics appears to be fairly ad hoc and consist in syntactically recognising certain HDL constructs and then rewriting them into the internal modelling language of the tool.

There has been some impressive and quite formal work on compiling Verilog to automata (e.g. (2)), but this does not attempt to formally justify the correctness of the translation with respect to the official (or de facto) simulation semantics. Rather it is an attempt to get new and improved semantics for better simulation and synthesis.

4. A SIMPLE HDL

The language described below is intended to embody the essence of the core of synthesisable Verilog, but with a lighter syntax.

A program consists of a set of concurrently executing non-terminating threads:

$$\begin{array}{l}\texttt{forever}\ \mathcal{S}_1\\ \texttt{forever}\ \mathcal{S}_2\\ \quad\vdots\\ \texttt{forever}\ \mathcal{S}_n\end{array}$$

where $\mathcal{S}_1$, $\mathcal{S}_2$, ... , $\mathcal{S}_n$ are *statements*, which are built out of *expressions* and *event controls*.

Each thread[2] models a piece of hardware, with interconnection specified by coincidence of variable names.

4.1. Expressions

Expressions are built up from variables and constants using operators (unary, binary, conditional etc.). The metavariables $\mathcal{V}$ and $\mathcal{E}$ will range over variables and expressions, respectively.

The structure of expressions is not specified in detail here. It is assumed that there are two Boolean constants `1` (*true* or *high*) and `0` (*false* or *low*), an infixed binary equality operator `=` and a conditional $\mathcal{E}\texttt{?}\mathcal{E}_1\texttt{:}\mathcal{E}_2$ (*if* $\mathcal{E}$ *then* $\mathcal{E}_1$ *else* $\mathcal{E}_2$). In examples, the C-like expression syntax of Verilog will be used.

It is assumed that each $\mathcal{E}$ expression has a *value* with respect to an assignment of values to each variable occurring in it. If s is such an assignment of values to variables, then the value of $\mathcal{E}$ is denoted by $[\![\mathcal{E}]\!]s$.

4.2. Event expressions

Event expressions $\mathcal{T}$ only occur as components of event controls $\texttt{@(}\mathcal{T}\texttt{)}$. They can be used both to delimit cycle boundaries and to specify combinational logic.

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

4.3. Statements

The syntax of statements $\mathcal{S}$ is given by the BNF-style grammar below.

$$\begin{array}{lcll}\mathcal{S} & ::= & \mathcal{V}_1,\ldots,\mathcal{V}_n\ \texttt{:=}\ \mathcal{E}_1,\ldots,\mathcal{E}_n & \text{(Assignment)}\\ & | & \mathcal{S}_1\texttt{;}\ \cdots\ \texttt{;}\ \mathcal{S}_n & \text{(Sequencing block)}\\ & | & \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \{\texttt{else}\ \mathcal{S}_2\} & \text{(Conditional)}\\ & | & \texttt{@(}\mathcal{T}\texttt{)}\ \mathcal{S} & \text{(Event control)}\end{array}$$

In an assignment $\mathcal{V}_1,\ldots,\mathcal{V}_n\ \texttt{:=}\ \mathcal{E}_1,\ldots,\mathcal{E}_n$, the variables $\mathcal{V}_1$, ... , $\mathcal{V}_n$ must be distinct. Each pair $(\mathcal{V}_i,\mathcal{E}_i)$ is called an *update*. The assignment is executed by updating in parallel each $\mathcal{V}_i$ with the value of $\mathcal{E}_i$.

Sometimes the notation $\overline{\mathcal{V}}$ will be used for the vector of variables $\mathcal{V}_1$, ... , $\mathcal{V}_n$ and $\overline{\mathcal{E}}$ for the vector of expressions $\mathcal{E}_1$, ... , $\mathcal{E}_n$.

[2]Threads correspond to `always`-blocks in Verilog.

4.4. Abbreviations

The following constructs are derived in terms of the syntax given above.

4.4.1. Case statements

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

abbreviates

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

4.4.2. Continuous assignments

assign $\overline{\mathcal{V}}$ = $\overline{\mathcal{E}}$

abbreviates

forever @($\mathcal{V}_1$ or $\cdots$ or $\mathcal{V}_r$) $\overline{\mathcal{V}}$:= $\overline{\mathcal{E}}$

where $\mathcal{V}_1, \ldots, \mathcal{V}_r$ are the variables occurring in any component expression of the vector $\overline{\mathcal{E}}$.

Verilog continous assignments have inertial delay, but the ones here exhibit transport delay. This difference only becomes apparent in the presence of #-delays (5), which are not considered here.

5. SEMANTIC PSEUDO-CODE

To simplify the presentation of the semantics, each thread is first compiled to a sequence of assembler-like pseudo instructions.

5.1. Pseudo-code instructions

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

$\overline{\mathcal{V}}$:= $\overline{\mathcal{E}}$	assignment
@($\mathcal{T}$)	event control
go n	unconditional jump to instruction n
ifnot $\mathcal{E}$ go n	jump to instruction n if $\mathcal{E}$ is not true

5.2. The size of a statement

The size function defined in this section is used in the translation algorithm described in 5.3. 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.

$$
\begin{aligned}
|\overline{\mathcal{V}}\ \texttt{:=}\ \overline{\mathcal{E}}| &= 1\\
|\mathcal{S}_1\texttt{;}\cdots\texttt{;}\ \mathcal{S}_n| &= |\mathcal{S}_1| + \cdots + |\mathcal{S}_n|\\
|\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}| &= |\mathcal{S}|+1\\
|\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2| &= |\mathcal{S}_1|+|\mathcal{S}_2|+2\\
|\texttt{@(}\mathcal{T}\texttt{)}| &= 1
\end{aligned}
$$

5.3. 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 $\langle\!\langle\mathcal{S}\rangle\!\rangle p$, 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). The symbol $\frown$ denotes sequence concatenation.

$$
\begin{aligned}
\langle\!\langle\overline{\mathcal{V}}\ \texttt{:=}\ \overline{\mathcal{E}}\rangle\!\rangle p &= \langle\overline{\mathcal{V}}\ \texttt{:=}\ \overline{\mathcal{E}}\rangle\\
\langle\!\langle\mathcal{S}_1\texttt{;}\ \mathcal{S}_2\texttt{;}\ldots\texttt{;}\mathcal{S}_n\rangle\!\rangle p &\\
&= \langle\!\langle\mathcal{S}_1\rangle\!\rangle p \frown \langle\!\langle\mathcal{S}_2\texttt{;}\ldots\texttt{;}\mathcal{S}_n\rangle\!\rangle(p+|\mathcal{S}_1|)\\
\langle\!\langle\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\rangle\!\rangle p &\\
&= \langle\texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ p+|\mathcal{S}|+1\rangle \frown \langle\!\langle\mathcal{S}\rangle\!\rangle(p+1)\\
\langle\!\langle\texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2\rangle\!\rangle p &= \langle\texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ p+|\mathcal{S}_1|+2\rangle\\
&\quad \frown \langle\!\langle\mathcal{S}_1\rangle\!\rangle(p+1)\\
&\quad \frown \langle\texttt{go}\ p+|\mathcal{S}_1|+|\mathcal{S}_2|+2\rangle\\
&\quad \frown \langle\!\langle\mathcal{S}_2\rangle\!\rangle(p+|\mathcal{S}_1|+2)\\
\langle\!\langle\texttt{@(}\mathcal{T}\texttt{)}\ \mathcal{S}\rangle\!\rangle\ p &= \langle\texttt{@(}\mathcal{T}\texttt{)}\rangle \frown \langle\!\langle\mathcal{S}\rangle\!\rangle(p+1)
\end{aligned}
$$

A thread forever S is translated into the infinite loop:

$\langle\!\langle\mathcal{S}\rangle\!\rangle 0 \frown \langle\texttt{go}\ 0\rangle$

5.4. Example translations

First an example to illustrate the translation of conditionals.

```
if ℰ
 then
  a := b;
  @(posedge clk) b := a;
 else
  a := b;
  b := a;


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

forever $\mathcal{S}$ compiles to a loop.

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


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

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 (12).

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


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

An explicit style of description of the machine above is given next. This is Example 8-17 from the Synopsys HDL Compiler for Verilog Reference Manual (12).

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


 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
```

6. EVENT SEMANTICS

The event semantics is specified via a simulation cycle that non-deterministically executes threads according to the event controls present.

Consider the following program:

forever $\mathcal{S}_1$
forever $\mathcal{S}_2$
⋮
forever $\mathcal{S}_n$

An *input* is a variable that occurs in one of $\mathcal{S}_1, \ldots, \mathcal{S}_n$, but does not occur on the left hand side of an assignment.

A *state variable* is a variable that occurs on the left hand side of an assignment. The values of the state variables are the *outputs* of the program.

It is assumed that each input 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 increases, 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 be enabled for execution. The simulation cycle (see 6 below) consists of repeatedly choosing an enabled thread, executing it until an event control is reached (an atomic step), and then enabling any new threads that fire. If several threads are simultaneously enabled, then the choice of which thread to advance is non-deterministic. If a state is reached in which there are no enabled threads, then the simulation cycle is said to have *quiesced*. When this happens the simulation time is incremented, the next set of inputs is assigned to the input variables and the whole process repeats.

Thus sequences of values on the inputs non-deterministically generate sequences of states (and hence a sequence of outputs). Such a sequence is called a *trace* and has the form:

$$\langle i_0, s_0\rangle\langle i_1, s_1\rangle\langle i_2, s_2\rangle \;\cdots$$

where s_0 is an initial state and $i_0 i_1 i_2 \ldots$ are inputs.

The state of a thread during simulation consists of:

- the simulation time (a non-negative integer);
- the values of all variables;
- the value of the program counter (denoted by *pc*).

It is assumed that the environment to the program supplies a value for each input at each instant of simulation time.

The simulation algorithm can be succinctly described if some terminology is introduced.

The instruction pointed to by the program counter of a thread is called the *current instruction*.

A thread is *enabled* if its current instruction is not an event control `@(`$\mathcal{T}$`)`. Enabled threads can be executed immediately. Once a thread starts executing it continues until an event control is reached. This execution is atomic and is called *advancing* the thread (see below).

A thread is *listening* if its current instruction is an event control `@(`$\mathcal{T}$`)`. Note that if a thread is not enabled then it is listening.

When a variable or input changes, an event expression may be *triggered*:

- $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ differs from the previous one;
- `posedge` $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ is `1` and the previous value was not `1`;
- `negedge` $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ is `0` and the previous value was not `0`;
- $\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$ is triggered if any of the $\mathcal{T}_i$ are triggered ($1 \leq i \leq n$).

A thread is *triggered* if it is listening and the event control of the current instruction is triggered. A triggered thread is *fired* by incrementing its program counter.

If a thread is not executing, but is enabled, then its current instruction cannot be an unconditional jump, but must be an assignment or a conditional jump. This is because, by the definition of $\langle\!\langle \mathcal{S} \rangle\!\rangle$, unconditional jumps cannot occur immediately after event controls, so when a thread is triggered the current instruction cannot become an unconditional jump.

Threads are executed concurrently. When an enabled thread is run it advances without interruption until an event control is reached. Thus a thread never stops running at an unconditional jump `go` n. To see the need for this consider:

<u>Thread 1</u>	<u>Thread 2</u>
`forever @(inp)` `   w := !inp;`	`forever @(w or inp)` `   outp := w & inp;`

`0: @(inp)`
`1: w := !inp`
`2: go 0`
	`0: @(w or inp)`
	`1: outp := w & inp`
	`2: go 0`

If Thread 2 could stop at instruction 2, then Thread 1 could interleave at that point and update to `w` whilst Thread 2 is not listening. By forcing all jumps to be taken immediately, this possibility is precluded.

The execution of a single instruction is performed as follows:

Instruction Execution

- $\mathcal{V}_1, \ldots, \mathcal{V}_n$ `:=` $\mathcal{E}_1, \ldots, \mathcal{E}_n$
  For $1 \leq i \leq n$ change the $\mathcal{V}_i$ component of the state to the value of $\mathcal{E}_i$ in the current state, note all variables whose value changes, then increment the program counter;
- `go` n
  Set the program counter to n;
- `ifnot` $\mathcal{E}$ `go` n
  If $\mathcal{E}$ is true in the current state then increment the program counter, otherwise set it to n.

When a thread fires these operations are repeated until an event control is reached and then any triggered threads (including possibly the one just executed) are fired. This process is called advancing a thread.

Advancing a Thread

- A thread is advanced by repeatedly executing the current instruction until an event control is reached.
- This execution is considered to be an atomic step and is not interruptable.
- As a thread is advancing, the variables that are modified by assignments are noted.
- When the thread stops advancing any listening threads that are guarded by event controls containing modified variables are fired.

Note that when a thread stops advancing it will be in a state whose current instruction is an event control. Such states are called *stopping states*. When reasoning about the execution of programs, it is sufficient to just consider the transitions of threads between stopping states, since these are the uninterruptible atomic actions of the thread. The decomposition of these actions into smaller ones, like unconditional jumps, is just an artifact of the way the event semantics is presented.

Inititially simulation time is 0, each program counter is set to 0 and each variable has some initial value (which is not specified here, but would typically be an 'undefined' value, like `x` in Verilog). The simulation algorithm consists of iterating forever the simulation step shown in the box below.

> **Simulation Step**
>
> - **There is an enabled thread:** Non-deterministically choose an enabled thread and advance it.
> - **No enabled threads – i.e. all threads listening:** Increment the simulation time, read in new input values and fire any triggered threads.

7. TRACE SEMANTICS

For each program, initial state and sequence of inputs, the simulation algorithm determines a set of traces. Each member of such a trace consists of the inputs at each time together with a state the simulation cycle quiesces to. Note that since simulation is non-deterministic, there may be several possible quiescent states (or none). Thus in general time is branching. However, in many special cases there is a unique quiescent state and so for these cases time can be modelled as linear. This is the case with the combinational and RTL programs analysed in sections 7.1 and 7.4 below. We show for these examples that the simulation algorithm, which is hard to work with directly, is in fact equivalent to the trace behaviour that is normally just assumed.

7.1. Combinational programs

The idea of a combinational program is that it computes its outputs instantaneously from its inputs. A syntactic condition that ensures this will be defined. In preparation, consider a thread having the form `forever @(`$\mathcal{T}$`)`$\mathcal{S}$, where $\mathcal{S}$ contains no event controls. This translates to pseudo code of the form:

0: `@(`$\mathcal{T}$`)`
$\vdots$
n: `go 0`

where the pseudo instructions `1`, `2`, ... , $n-1$ are assignments or jumps (i.e. not event controls).

Since threads execute atomically from event control to event control, it is clear that such a pseudo program is equivalent to one of the form:

0: `@(`$\mathcal{T}$`)`
1: $\overline{\mathcal{V}}$ `:=` $\overline{\mathcal{E}}$
2: `go 0`

For example,

```
forever @(y or z)
  x := 1;
  if x>y then x,y := y,x else y,z := 3,y
  z := x+y
```

translates to:

```
0: @(y or z)
1: x := 1
2: ifnot x>y go 5
3: x,y := y,x
4: go 6
5: y,z := 3,y
6: z := x+y
7: go 0
```

which is equivalent to:

```
0: @(y or z)
1: x,y,z :=
     x>y ? y : 1,
     x>y ? 1 : 3,
     (x>y ? y : 1) + (x>y ? 1 : 3)
2: go 0
```

A formal description of the straightforward conversion of combinational threads into a single parallel assignment, as illustrated by this example, is not given here.

A *weakly combinational* program has the form:

`forever @(`$\mathcal{T}_1$`)` $\overline{\mathcal{V}}_1$`:=`$\overline{\mathcal{E}}_1$
$\vdots$
`forever @(`$\mathcal{T}_n$`)` $\overline{\mathcal{V}}_n$`:=`$\overline{\mathcal{E}}_n$

and satisfies three conditions:

1. $\mathcal{T}_i$ consists of all the variables occurring in $\overline{\mathcal{E}}_i$ `or`-ed together;
2. no variable occurs more than once on the left of an assignment (no clashing assignments);
3. after performing any update $(\mathcal{V}, \mathcal{E})$, the value of $\mathcal{V}$ in the resulting state equals the value of $\mathcal{E}$.

Updates satisfying condition 3 above are called idempotent. More formally, $(\mathcal{V}, \mathcal{E})$ is *idempotent* if

$$[\![\mathcal{V}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) = [\![\mathcal{E}]\!]s = [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

where s maps variables to their values and $s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]$ denotes the mapping identical to s except that the value of variable $\mathcal{V}$ is the value of $\mathcal{E}$.

A sufficient condition that $(\mathcal{V}, \mathcal{E})$ be idempotent is that $\mathcal{V}$ not occur in $\mathcal{E}$.

If the simulation cycle of a weakly combinational program quiesces then it does so with a unique value for all the variables. Showing quiescence requires an additional condition (loop-free) which is defined later.

A weakly combinational program generates n threads:

Thread 1			**Thread n**	
0:	`@(`$\mathcal{T}_1$`)`		0:	`@(`$\mathcal{T}_n$`)`
1:	$\overline{\mathcal{V}}_1$ `:=` $\overline{\mathcal{E}}_1$	$\cdots$	1:	$\overline{\mathcal{V}}_n$ `:=` $\overline{\mathcal{E}}_n$
2:	`go 0`		2:	`go 0`

7.1.1. *An invariant property*

Suppose thread i has program counter pc_i. Let *Inv* be the n-ary conjunction $\bigwedge_{i=1}^{n} Inv_i$, where Inv_i is the statement:

$$(pc_i = 0 \Rightarrow \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i) \wedge (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i \Rightarrow pc_i = 1)$$

Inv_i asserts two things: (i) if the i-th thread is at instruction 0 then the value of the vector of variables $\overline{\mathcal{V}}_i$ equals the value of the vector of expressions $\overline{\mathcal{E}}_i$ and (ii) if any of these values are not equal then the thread is at instruction 1.

Note that *Inv* is a predicate on states. This could be made explicit, by defining $Inv(s)$ as follows:

$$\bigwedge_{i=1}^{n} \begin{array}{l} (\llbracket pc_i \rrbracket s = 0 \Rightarrow \llbracket \overline{\mathcal{V}}_i \rrbracket s = \llbracket \overline{\mathcal{E}}_i \rrbracket s) \\ \wedge \\ (\llbracket \overline{\mathcal{V}}_i \rrbracket s \neq \llbracket \overline{\mathcal{E}}_i \rrbracket s \Rightarrow \llbracket pc_i \rrbracket s = 1) \end{array}$$

However, where the state is clear from context the less formal implicit-state notation will be used.

Suppose *Inv* holds. There are two cases to consider: there are some enabled threads, or there are no enabled threads.

1. If there are some enabled threads, then one of them, say thread i, is chosen and executed. Since all updates are idempotent, the execution will cause the value of $\overline{\mathcal{V}}_i$ to become equal the value of $\overline{\mathcal{E}}_i$, hence Inv_i will hold immediately after the assignment is executed. Next, other threads may be fired (including, possibly, thread i itself). The following case analysis established *Inv* continues to hold after this.
   (a) If thread j is enabled but not selected for execution, then $pc_j = 1$, so Inv_j remains vacuously true.
   (b) If a thread j is triggered by the execution of thread i (the case $j{=}i$ is possible), then thread j becomes enabled with $pc_j = 1$ and so Inv_j becomes vacuously true.
   (c) If a thread j is listening but not triggered then it must be the case that no variable changed by executing thread i occurs in $\overline{\mathcal{E}}_j$, so the value of $\overline{\mathcal{E}}_j$ is unchanged. Also pc_j remains equal to 0.
      i. If $j = i$ then as all updates are idempotent the execution of $\overline{\mathcal{V}}_i$:= $\overline{\mathcal{E}}_i$ makes the value of each variable in $\overline{\mathcal{V}}_i$ equal to the value of the corresponding expression in $\overline{\mathcal{E}}_i$ and hence Inv_i holds.
      ii. If $j \neq i$, the variables in $\overline{\mathcal{V}}_j$ are distinct from those in $\overline{\mathcal{V}}_i$, so the value of $\overline{\mathcal{V}}_j$ is unchanged and thus Inv_j remains true.
2. If there are no enabled threads, then time will advance and if the inputs changes some threads may be fired. If no threads are fired, then the values of all variables occurring in any $\overline{\mathcal{E}}_i$ $(1 \leq i \leq n)$ will be unchanged. Since the environment only changes inputs, the value of $\overline{\mathcal{V}}_i$ is unchanged. Thus *Inv* continues to hold.
   If thread i is fired then $pc_i = 1$ and Inv_i becomes vacuously true.

Thus *Inv* is an invariant of each simulation step. If simulation quiesces, then all program counters will be 0 and so for each i the value of $\overline{\mathcal{V}}_i$ will equal the value of $\overline{\mathcal{E}}_i$.

7.1.2. A termination property

To establish that the simulation of combinational programs always quiesces – i.e. doesn't go into an infinite loop without advancing time – it is sufficient to exhibit a 'variant' that decreases on each iteration of the simulation algorithm for which there is an enabled thread. Such a variant shows that eventually a simulation state must be reached in which there are no enabled threads, and hence time advances. A suitable variant uses the Dershowitz-Manna multiset ordering (3). Under this ordering multiset M_2 is less than multiset M_1 if M_2 can be obtained from M_1 by removing a finite number of elements and replacing each of them by a finite number of strictly smaller elements. This ordering is well-founded if the ordering on the elements is.

The idea of the termination proof is to show that each chain of events – one causing the next – is finite. Define $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$ to mean, roughly, that doing the update $(\mathcal{V},\mathcal{E})$ will immediately trigger a thread containing the update $(\mathcal{V}',\mathcal{E}')$.

$(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}') \equiv_{Def} \exists s. \llbracket \mathcal{E}' \rrbracket s \neq \llbracket \mathcal{E}' \rrbracket (s[\mathcal{V} \leftarrow \llbracket \mathcal{E} \rrbracket s])$

The relation $\nearrow$ is defined semantically, but for weakly combinational programs it implies a syntactic relation $\boldsymbol{\nearrow}$ defined by:

$(\mathcal{V},\mathcal{E}) \boldsymbol{\nearrow} (\mathcal{V}',\mathcal{E}') \equiv_{Def} \mathcal{V}$ *occurs in* $\mathcal{E}'$ *and* $\mathcal{V} \neq \mathcal{V}'$

If $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$ then $(\mathcal{V},\mathcal{E}) \boldsymbol{\nearrow} (\mathcal{V}',\mathcal{E}')$. Assume that $(\mathcal{V},\mathcal{E}) \boldsymbol{\nearrow} (\mathcal{V}',\mathcal{E}')$ doesn't hold, then there are two cases:

Case 1: $\mathcal{V}$ does not occur in $\mathcal{E}'$.
In this case it is clear that for all s:

$$\llbracket \mathcal{E}' \rrbracket s \;=\; \llbracket \mathcal{E}' \rrbracket (s[\mathcal{V} \leftarrow \llbracket \mathcal{E} \rrbracket s])$$

which contradicts

$$\exists s. \quad \llbracket \mathcal{E}' \rrbracket s \neq \llbracket \mathcal{E}' \rrbracket (s[\mathcal{V} \leftarrow \llbracket \mathcal{E} \rrbracket s])$$

hence it can't be the case that $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$.

Case 2: $\mathcal{V} = \mathcal{V}'$.
There are no clashing assignments, thus $\mathcal{E} = \mathcal{E}'$, so if $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$ then $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V},\mathcal{E})$, i.e. $\exists s. \llbracket \mathcal{E} \rrbracket s \neq \llbracket \mathcal{E} \rrbracket (s[\mathcal{V} \leftarrow \llbracket \mathcal{E} \rrbracket s])$, which contradicts the requirement that $(\mathcal{V},\mathcal{E})$ be idempotent.

In both cases it follows that it is not the case that $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$.

Thus by contradiction $(\mathcal{V},\mathcal{E}) \nearrow (\mathcal{V}',\mathcal{E}')$ entails $(\mathcal{V},\mathcal{E}) \boldsymbol{\nearrow} (\mathcal{V}',\mathcal{E}')$.

Define a $\nearrow$*-chain* in a set of assignments to be a sequence $(\mathcal{V}_1,\mathcal{E}_1), \ldots, (\mathcal{V}_l,\mathcal{E}_l)$ of updates such that for $1 \leq p < l$:

(i) $\mathcal{V}_p$ is updated to $\mathcal{E}_p$ by one of the assignments in the set;
(ii) $(\mathcal{V}_p,\mathcal{E}_p) \nearrow (\mathcal{V}_{p+1},\mathcal{E}_{p+1})$.

Call a set of assignments *loop-free* if the lengths of all $\nearrow$-chains are bounded.

Define a program to be *combinational* if it is weakly combinational and loop free.

Note that since every $\nearrow$-chain is also a $\nearrow$-chain, if there is an infinite $\nearrow$-chain then there is also an infinite $\nearrow$-chain. So checking that all $\nearrow$-chains are finite ensures the program is loop-free and hence combinational.

Define the *height* of $\mathcal{V}$ to be the length of the longest $\nearrow$-chain in the assignments $\overline{\mathcal{V}}_1$:=$\overline{\mathcal{E}}_1$, ... , $\overline{\mathcal{V}}_n$:=$\overline{\mathcal{E}}_n$, whose first element is $(\mathcal{V}, \mathcal{E})$, for some $\mathcal{E}$.

To show each simulation cycle quiesces, consider the sequence of steps that happen after the advance of simulation time. If no inputs change then the cycle is already quiescent and there is nothing to show. If some inputs change, then some threads will fire and become enabled with their program counters at 1.

The subsequent quiescence (termination) of the simulation algorithm can be verified by taking as variant the multiset of the heights of those variables $\mathcal{V}$ such that the value of $\mathcal{V}$ differs from the value of $\mathcal{E}$. Note that by *Inv* all such updates must be part of an enabled thread.

Suppose some threads are enabled and one of them, thread i say, is selected for execution. All the variables in $\overline{\mathcal{V}}_i$ are updated to the corresponding values $\overline{\mathcal{E}}_i$, and the thread advances to instruction 0. Next, every thread j such that $\mathcal{T}_j$ contains a variable modified by executing thread i is fired and so becomes enabled.

If an update $(\mathcal{V}, \mathcal{E})$ was done by executing thread i then, since it is idempotent, the value of $\mathcal{V}$ will equal the value of $\mathcal{E}$ after the update and so the height of $\mathcal{V}$ will be removed from the multiset. This deleted height will be replaced with the heights of other variables $\mathcal{V}'$ such that there is an update $(\mathcal{V}', \mathcal{E}')$ and the value of $\mathcal{V}'$ becomes different from the value of $\mathcal{E}'$. Suppose s is the state in which $(\mathcal{V}, \mathcal{E})$ was done. The height of $\mathcal{V}'$ will be added if $[\![\mathcal{V}']\!]s = [\![\mathcal{E}']\!]s$ but:

$$[\![\mathcal{V}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

There are two cases to consider:

Case 1: If $\mathcal{V} \neq \mathcal{V}'$ this equation is:

$$[\![\mathcal{V}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

i.e.

$$[\![\mathcal{E}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

hence $(\mathcal{V}, \mathcal{E}) \nearrow (\mathcal{V}', \mathcal{E}')$.

Case 2: If $\mathcal{V} = \mathcal{V}'$ then (as each variable can only be on the left of at most one update) $\mathcal{E} = \mathcal{E}'$ and hence:

$$[\![\mathcal{V}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) \neq [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

i.e.

$$[\![\mathcal{E}]\!]s \neq [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

but this is impossible as $(\mathcal{V}, \mathcal{E})$ is assumed idempotent.

Thus each added height – i.e. the heights of each such $\mathcal{V}'$ – will be less than the height of $\mathcal{V}$, so the multiset decreases in the Dershowitz-Manna ordering.

Each step of the simulation algorithm decreases the multiset. As long as there are enabled threads the simulation cycle continues, so by the invariant *Inv* and the well-foundedness of the multiset ordering eventually there must be no enabled threads, i.e. the simulation quiesces.

7.2. Transparent latches

Consider:

```
forever @(clk or d) if clk then q := d
```

If this is converted to an equivalent assignment, the result is:

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

The update (q, clk ? d : q) is idempotent, since clearly:

$$\forall s.\ [\![\texttt{clk ? d : q}]\!]s = [\![\texttt{clk ? d : q}]\!](s[\texttt{q} \leftarrow [\![\texttt{clk ? d : q}]\!]s])$$

It is not the case that:

$$(\texttt{q}, \texttt{clk ? d : q}) \nearrow (\texttt{q}, \texttt{clk ? d : q})$$

hence the latch is loop-free.

Define an update $(\mathcal{V}, \mathcal{E})$ to be *latch-like* if $\mathcal{E}$ is either $\mathcal{V}$ or a nested conditional expression in which the only occurrences of $\mathcal{V}$ are in the `then` or `else` part of conditionals.

Define a thread `forever @(`$\mathcal{T}$`)` $\overline{\mathcal{V}}$:=$\overline{\mathcal{E}}$ to be latch-like if every update $(\mathcal{V}, \mathcal{E})$ in the assignment is either latch-like or has $\mathcal{V}$ not occurring in $\mathcal{E}$. Note that all such updates will be idempotent.

Consider a weakly combinational program:

forever @($\mathcal{T}_1$) $\overline{\mathcal{V}}_1$:=$\overline{\mathcal{E}}_1$
⋮
forever @($\mathcal{T}_n$) $\overline{\mathcal{V}}_n$:=$\overline{\mathcal{E}}_n$

where each thread is latch-like. A sufficient condition for this to be combinational is that every chain:

$$(\mathcal{V}_1, \mathcal{E}_1) \nearrow (\mathcal{V}_2, \mathcal{E}_2) \nearrow (\mathcal{V}_3, \mathcal{E}_3)\ \ldots$$

is finite.

The fact that loop-free latch-like threads are combinational shows that the use here of the term is a bit non-standard. One might expect that "combinational" would denote circuits with no memory.

7.2.1. Summary

If $\mathcal{S}_1$, ... , $\mathcal{S}_n$ contain no event controls then any program of the form:

forever @($\mathcal{T}_1$) $\mathcal{S}_1$ $\cdots$ forever @($\mathcal{T}_n$) $\mathcal{S}_n$

is equivalent to a program of the form:

forever @($\mathcal{T}_1$) $\overline{\mathcal{V}}_1$:=$\overline{\mathcal{E}}_1$
⋮
forever @($\mathcal{T}_n$) $\overline{\mathcal{V}}_n$:=$\overline{\mathcal{E}}_n$

Such a program is combinational if it is weakly combinational and loop-free. A sufficient condition is that every thread is latch-like and there are no infinite $\nearrow$-chains.

Each simulation step of a combinational program preserves the invariant:

$$\bigwedge_{i=1}^{n} \ (pc_i = 0 \ \Rightarrow \ \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i) \\ \wedge \ (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i \ \Rightarrow \ pc_i = 1)$$

and as long as inputs do not change, each step reduces the size of the variant and thus the simulation cycle must quiesce with all threads listening at pseudo-instruction 0 and hence with $\overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i$.

Thus if s_0 is an initial state and $i_0 i_1 i_2 \ldots$ are the inputs, then the traces of this combinational program are sequences:

$$\langle i_0, s_0\rangle\langle i_1, s_1\rangle\langle i_2, s_2\rangle \ \cdots$$

where the value of each variable in $\overline{\mathcal{V}}_i$ in state s_{t+1} is the value of the corresponding expression in $\overline{\mathcal{E}}_i$ evaluated with inputs i_{t+1} and state s_t.

7.3. Flip-flops

A flip-flop is a single register with input `d` and output `q` that is clocked on the positive edge of a clock `clk` (which is assumed distinct from `d` and `q`). This is modelled by:

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

It is clear that the effect of the simulation algorithm is to update the value of `q` with the value input on `d` at each simulation time that triggers `posedge clk` (i.e. at each time that `clk` is high, but was not high at the preceding time).

If $f(t)$ denotes the value of input or variable f at simulation time t, then the trace semantics of the register above is:

$$\mathtt{q}(t{+}1) \ = \ \mathit{if} \ \mathtt{clk}(t{+}1) = 1 \ \wedge \ \mathtt{clk}(t) \neq 1 \\ \mathit{then} \ \mathtt{d}(t) \ \mathit{else} \ \mathtt{q}(t)$$

Define:

$$(\mathtt{posedge} \ \mathit{clk})(t+1) \ \equiv_{\mathit{Def}} \\ \mathtt{clk}(t{+}1) = 1 \ \wedge \ \mathtt{clk}(t) \neq 1$$

then the trace semantics can be written:

$$\mathtt{q}(t{+}1) \ = \ \mathit{if} \ (\mathtt{posedge\ clk})(t+1) \\ \mathit{then} \ \mathtt{d}(t) \ \mathit{else} \ \mathtt{q}(t)$$

A bank of n registers can be represented by a single parallel assignment:

```
forever @(posedge clk) q̅ := d̅
```

where $\overline{\mathtt{q}} = (\mathtt{q}_1, \ldots, \mathtt{q}_n)$ and $\overline{\mathtt{d}} = (\mathtt{q}_1, \ldots, \mathtt{d}_n)$ and $\mathtt{q}_i \neq \mathtt{q}_j$ if $i \neq j$. The trace semantics is:

$$\bigwedge_{i=1}^{n} \ \mathtt{q}_i(t{+}1) \ = \\ \mathit{if} \ (\mathtt{posedge\ clk})(t+1) \ \mathit{then} \ \mathtt{d}_i(t) \ \mathit{else} \ \mathtt{q}(t)$$

An alternative representation, with the same trace semantics, is to have n separate threads; one for each individual register.

```
forever @(posedge clk) q1 := d1
   ⋮
forever @(posedge clk) qn := dn
```

The trace equivalence of the two representations is assumed clear.

Consider now some combinational logic, say an inverter, connected to the input of a flip-flop. This could be represented with one thread by:

```
forever @(posedge clk) q := !d
```

or with the inverter and flip-flop in different threads:

```
forever @(d) a := !d
forever @(posedge clk) q := a
```

If `d` changes at the same time as a positive edge on `clk` then the second representation, but not the first, might store the negation of the unchanged (i.e. previous) value of `d`. This would happen if the flip-flop thread was executed first. The subsequent execution of the inverter thread will set `a` to the negation of the new value of `d`, but since the flip-flop is not sensitive to changes in `a` it won't update the value of `q`.

The trace semantics of the single thread version is:

$$\mathtt{q}(t{+}1) \ = \ \mathit{if} \ (\mathtt{posedge\ clk})(t+1) \\ \mathit{then} \ \mathtt{!d}(t) \ \mathit{else} \ \mathtt{q}(t)$$

and of the two threads is:

$$(\forall\, t.\ \mathtt{a}(t) \ = \ \mathtt{!d}(t)) \\ \wedge \\ (\forall\, t.\ \mathtt{q}(t{+}1) \ = \ \mathit{if} \ (\mathtt{posedge\ clk})(t+1) \\ \mathit{then} \ \mathtt{a}(t) \ \mathit{else} \ \mathtt{q}(t))$$

If it is assumed that inputs never change on clock edges[3] then the two representations produce the same signal trace on `q`.

If the combinational logic is connected to the output of the flip-flop the problem doesn't arise. Thus no matter what order the threads in:

```
forever @(posedge clk) a := d
forever @(a) q := !a
```

are executed, the final quiescent state will be the same – namely `q` will equal the negation of the value of `d`.

$$(\forall\, t.\ \mathtt{a}(t{+}1) \ = \ \mathit{if} \ (\mathtt{posedge\ clk})(t+1) \\ \mathit{then} \ \mathtt{d}(t) \ \mathit{else} \ \mathtt{a}(t)) \\ \wedge \\ (\forall\, t.\ \mathtt{q}(t) \ = \ \mathtt{!a}(t))$$

[3] This assumption on the environment can be hard or impossible to enforce if the inputs are asynchronous – however clever analogue engineering can make it true with high probablility.

However, with this example there is a hazard: if the inverter executes first then `q` will be set to the negation of the old value of `a`. The subsequent execution of the flip-flop will update `a` and might then change the value of `q`, since the inverter is sensitive to changes in `a`. The hazard is only visible at the event semantics level; it is invisible with trace semantics.

7.4. RTL programs

Register Transfer Level (RTL) hardware consists of combinational logic and flip-flops.

An HDL representation of this is a program that can be partitioned into a set of threads that are a combinational program plus a set of flip-flops of the form:

$$\texttt{forever @(posedge clk) } \overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}}$$

It is assumed that there is a single input clock `clk` and that the environment is such that no other inputs change when `clk` has a positive edge. Also it is assumed no two distinct flip-flops drive the same ourput (no clashing assignments).

Thus an RTL program has the form:

$$\begin{array}{l}
\texttt{forever @(}\mathcal{T}_1\texttt{) } \overline{\mathcal{V}}_1 \texttt{ := } \overline{\mathcal{E}}_1 \\
\quad \vdots \\
\texttt{forever @(}\mathcal{T}_m\texttt{) } \overline{\mathcal{V}}_m \texttt{ := } \overline{\mathcal{E}}_m \\
\texttt{forever @(posedge clk) } \overline{\mathcal{V}}_{m+1} \texttt{ := } \overline{\mathcal{E}}_{m+1} \\
\quad \vdots \\
\texttt{forever @(posedge clk) } \overline{\mathcal{V}}_{m+n} \texttt{ := } \overline{\mathcal{E}}_{m+n}
\end{array}$$

where the first m threads constitute a combinational program and the last n threads are flip-flops.

A simulation cycle that starts with a positive edge on `clk` will (by assumption) be one for which none of the other inputs change. Thus the positive edge will enable all the flip-flops and none of the other threads. After m simulation steps (i.e. one to execute each flip-flop) the output variables will have been updated and the cycle will quiesce.

A simulation cycle that doesn't start with a positive edge on `clk` will not enable any of the flip-flops. The rest of the threads will simulate as combinational logic and, by the analysis in 7.1, will quiesce with each variable set to the value of the expression that updates it. Thus the trace semantics of the RTL program above is:

$$\begin{array}{l}
(\forall t.\ \overline{\mathcal{V}}_1(t) \ = \ \overline{\mathcal{E}}_1(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_m(t) \ = \ \overline{\mathcal{E}}_m(t)) \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_{m+1}(t{+}1) \ = \ \mathit{if}\ (\texttt{posedge clk})(t+1) \\
\qquad\qquad\qquad\qquad \mathit{then}\ \overline{\mathcal{E}}_{m+1}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+1}(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall t.\ \overline{\mathcal{V}}_{m+n}(t{+}1) \ = \ \mathit{if}\ (\texttt{posedge clk})(t+1) \\
\qquad\qquad\qquad\qquad \mathit{then}\ \overline{\mathcal{E}}_{m+n}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+n}(t))
\end{array}$$

These equations are exactly how the RTL program would be represented for formal verification, thus we have confirmed that for RTL programs the event and trace semantics are in harmony.

8. CONCLUSIONS AND FUTURE WORK

This paper has shown that a simple but formal simulation cycle applied to reasonable combinations of flip-flops and combinational logic is consistent with the normal semantics used for formal verification.

A long term goal is to provide a single 'golden' formal semantics for Verilog from which other application-specific semantics can be mechanically derived.

ACKNOWLEDGEMENTS

This work was funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC) via grants GR/K57343 and GR/L74262/01. The principal Investigator of both these is Dr. David Greaves.

We are grateful to Synopsys Inc. for providing us with their software.

Thanks to Abhijit Ghosh, David Greaves, Daryl Stewart, Ted Stanion and Myra VanInwegen for discussions on the semantics of Verilog, and to the anonymous referees for pointing out errors and suggesting improvements.

REFERENCES

[1] Jonathan Bowen, He Jifeng, and Xu Qiwen. An animatable operational semantics of the verilog hardware description language. In Shaoying Liu, John A. McDermid, and Michael G. Hinchey, editors, *Proceedings of ICFEM2000: 3rd IEEE International Conference on Formal Engineering Methods*, pages 199–297. IEEE Computer Society Press, September 2000.

[2] Szu-Tsung Cheng. *Compilation, Synthesis, and Simulation of Hardware Description Languages – The Compositional Models of HDL's.* PhD thesis, University of California, Berkeley, 1998.

[3] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM (CACM)*, 22(8):465–476, August 1979.

[4] John Fiskio-Lasseter. A formal description of behavioral verilog based on axiomatic semantics. Master's thesis, University of Oregon, 1998. Tech. Rep. TR-98-04, University of Oregon, Computer and Information Science Department.

[5] 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.

[6] The VIS Group. Vis: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification, New Brunswick, NJ*, pages 428–432. Springer Lecture Notes in Computer Science 1102, July 1996.

[7] Zhu Huibiao and He Jifeng. A dc-based semantics for verilog. In David Notkin Yulin Feng and Marie-Claude Gaude, editors, *International Conference on Software: Theory and Practice (ICS2000)*, pages 421–432, August 2000. Available from `http://www.iist.unu.edu/newrh/III/1/page.html`.

[8] 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.

[9] D. Knapp, T. Ly, D. MacMillen, and R Miller. Behavioural synthesis methodology for HDL-based specification and validation. In *Proc. of 32nd DAC*, San Francisco, June 1995. ACM.

[10] 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.

[11] Gordon Pace. *Hardware Design Based on Verilog HDL*. PhD thesis, DPhil thesis, University of Oxford, August 1998.

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