An Animatable Operational Semantics of the Verilog Hardware Description Language

Jonathan P. Bowen
South Bank University
Centre for Applied Formal Methods
SCISM, Borough Road, London SE1 0AA, UK
Email: *jonathan.bowen@sbu.ac.uk*
URL: *http://www.jpbowen.com/*

He Jifeng & Xu Qiwen
The United Nations University
International Institute for Software Technology
UNU/IIST, P.O. Box 3058, Macau
Email: {*jifeng, qxu*}*@iist.unu.edu*
URL: *http://www.iist.unu.edu/*

Abstract

An operational semantics of a significant subset of the Verilog Hardware Description Language (HDL) is presented. The semantics is encoded using the logic programming language Prolog in a literate programming style. This allows the associated documentation to be maintained in step with the semantics, and the printed version to be presented in a standard mathematical operational semantics style. It also enables the semantics to be directly animated using a Prolog interpreter. Using this approach allows the exploration of sometimes subtle behaviours of parallel programs and the possibility of rapid changes or additions to the semantics of the language covered that could be missed otherwise. In addition, it provides an extra check on the validity of the operational semantics.

1. Introduction

The use of formal methods is as successful, if not more successful, in the realm of hardware design as it is in software [20, 21]. This paper investigates the formalization of a widely used Hardware Description Language (HDL), Verilog [11]. The use of the Prolog logic programming language [8] to encode this as an operational semantics [30, 33] allows the possibility of direct execution of this formal description.

An important feature of a specification is that it is not necessarily executable [4, 14] although some contend that it is helpful if it is [2, 10]. An animation of a specification can help in the understanding of it, just as formal reasoning can too, and in a complementary manner. Here we accept that a specification of a semantics is not normally directly executable, especially if it is desirable to include non-deterministic aspects in the description. However, if the non-determinism can be limited finitely, we may be able to model it successfully in a usefully executable way.

Non-determinism is particularly helpful in the formulation of the meaning of parallel programs since the exact ordering of interleaving of parallel execution of processes may not be known. In the simulation of hardware, this is very important since hardware systems are naturally parallel, consisting of a large number of components, all constantly "executing" concurrently.

Section 1.1 introduces two widely used HDLs, VHDL and Verilog, including formalization of these HDLs and section 1.2 covers the use of logic programming (specifically Prolog) for rapid-prototyping specifications with minimal development, especially when the specification includes non-deterministic aspects. Section 1.3 briefly introduces the literate programming style used in the formulation of the semantics presented in the rest of this paper. Section 2, the main part of the paper, presents a Verilog operational semantics originally encoded using Prolog and Section 3 gives some example animations using this semantic description.

1.1. Hardware Description Languages

Hardware Description Languages (HDLs) are an increasingly popular way to develop hardware in industry as tool support improves and standards become established. Two of the major HDLs in use are VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language) [38] and Verilog [11]. Both VHDL and Verilog have IEEE standards associated with them [23, 24]. For extensive online information relating to Verilog, see [1].

The formal semantics of VHDL has been studied quite extensively [9], but that of Verilog less so, even though VHDL is probably a more complex language than Verilog. A start to providing a formal semantics for Verilog has been made by Gordon [12, 13], but this covers a relatively small subset of the language. An Open Verilog International (OVI) [31] *Formal Verification Standards Working Group* has been established with an aim to ensure interoper-

3rd IEEE International Conference on Formal Engineering Methods (ICFEM'00)
0-7695-0822-7/00 $10.00 @ 2000 IEEE

ability among formal verification and other tools [35]. This is concentrating its effort on a synthesizable Verilog subset.

Combinational Verilog programs have been explored formally [37]. In addition, an operational semantics [30, 33] for a reasonably substantial subset of the language has been formulated [36]. Subsequently this semantics has been refined in a working paper [18]. An advantage of the availability of an operational semantics for a language is the increased understanding and the possibility of formal reasoning that this brings. A disadvantage is that the semantics cannot be directly animated. An operation semantics can allow non-determinism, which is advantageous in a specification, but potentially problematic in a simulator where one execution path must normally be selected.

1.2. Logic Programming

The logic programming language Prolog [8] is an excellent rapid-prototyping language which can be applied in many areas, including digital hardware circuits [7]. In particular, it allows the execution of relations in a possibly non-deterministic manner, which can be especially useful in the modelling of parallel systems. Once the relations given in a specification are encoded in Prolog, the execution proceeds using a simple depth-first left to right search, depending on the order of the clauses used for the encoding. It is important to constrain any non-determinism in a finite way if termination is to be ensured.

Prolog includes a pseudo-clause called op(...) that allows the definition of infix, prefix and postfix operators with a specified precedence and also left or right associativity if required. This facility, although not extensively used by many Prolog programmers, allows a Prolog program to follow the form of a mathematical definition consisting of a set of relations fairly directly. The encoding of the relations themselves is typically of the same order of size as the original relations. Inevitably, most mathematical relations include constraints on numbers, sets, etc. Prolog provides simple integer arithmetic and support for lists that can be used to encode sets in a simple manner. Normally some extra clauses are required for encoding the constraints, but typically these can be of the same order of magnitude in size as the original relations themselves.

1.3. Literate Programming

In developing the Prolog source program for the operational semantics presented in this paper, the "*literate programming*" style of Knuth [26, 27] was adopted. Most programming languages expect input to be in the form of source program except where portions of the input file are explicitly marked as comments in some way. In the literate programming style, the source file is considered as a document in its own right, containing portions of program only in explicitly marked sections. This allows documentation to be maintained in the same location (file) as the program itself, aiding the goal of consistency between them during program development.

Normally text between /*...*/ is considered to be a comment by most Prolog systems. Instead we view text between */.../* as sections of Prolog program. The rest is the associated documentation in the LaTeX document preparation system [28] source (ASCII) format used to generate the report on which this paper is based [3].

The entire contents of Sections 2 and 3 have been processed from the original Prolog source program and output generated by the Prolog simulator (which can be treated in a similar manner to Prolog code) with associated LaTeX format documentation in this manner and then included directly. The processing to do this has been fully automated. As the program was updated during development, so was the original documentation [3].

2. Verilog Simulator Semantics

In this section, the main part of this paper, we present an an operational semantics for a significant subset of Verilog encoded using the logic programming language Prolog.

2.1. Sequential features

For the standard sequential features of the language, only the program and state variable components are required in the specification of transition rules. Procedural assignment overrides the state value associated with a variable V (e.g., a wire or wires in a circuit) with the new value established by the expression E. The program is then terminated (indicated by ϵ).

$$\frac{\Sigma' = \Sigma \oplus (V{=}E)}{\langle V = E, \Sigma\rangle \rightarrow \langle \epsilon, \Sigma'\rangle.}$$

The "$\oplus$" overriding operation above replaces a variable value with a new expression in the state Σ to establish a new state (Σ' above) with the values of the rest of the variables unchanged. The formal encoding for "$\oplus$" may be found in [3]. The transition relation $\_ \rightarrow \_$ relates a sequential program and its associated state before the transition to a new sequential program and possibly updated state after the transition. The transition is assumed to be instantaneous.

The delay operator # allows a delay of N time units (typically clock cycles in hardware). There are two possibilities. Firstly the delay may terminate successfully and unconditionally:

$$\frac{\mathsf{true}}{\#N \text{-}\langle N\rangle\!\rightarrow \epsilon.}$$

The transition relation $\_ \text{-}\langle N\rangle\!\rightarrow \_$ relates a sequential program before the transition to a new sequential program after the transition, where the transition takes N time units.

Secondly (and non-deterministically), the program may execute for some number of delay units less than the maximum specified:

$$\frac{0 < N' < N \ \wedge \ T = N - N'}{\#N \text{-}\langle T \rangle\!\rightarrow \#N'.}$$

Note that Prolog cannot resolve arithmetic expressions to their values in parameters, unlike most other programming languages; hence the use of T above to hold the value of $N - N'$. We also explicitly constrain the delay to be greater than zero to ensure that progress is made. Otherwise time may "stand still" in the simulation and the program may never terminate as a result. If this is not an issue, we could relax this constraint and allow zero delays if required. The calculation of N' is encoded in [3] to return smaller values first so that maximal time progress is made if possible.

Events are another non-deterministic aspect of Verilog. There are three possibilities. Firstly the event may occur. This is determined from a change of state from Σ_1 to Σ_2 which must be provided to this clause:

$$\frac{\langle \Sigma_1, \Sigma_2 \rangle \ \models \ \mathit{Event}}{@\mathit{Event} \cdot S \text{-}\langle \Sigma_1, \Sigma_2 \rangle\!\rightarrow S.}$$

In the Verilog program $@\mathit{Event} \cdot S$, the sub-program S is only executed after *Event* has occurred. The transition relation $\_ \text{-}\langle \Sigma_1, \Sigma_2 \rangle\!\rightarrow\_$ relates a sequential program before the transition to a new sequential program after the transition, where the states Σ_1 and Σ_2 are used to determine if an event has occurred. $\langle \Sigma_1, \Sigma_2 \rangle \ \models \ \mathit{Event}$ means that the event *Event* (for example, the change of the value of one or more wires in a circuit) can be proved to have occurred from the change in state between Σ_1 and Σ_2. A formal definition can be found in [3].

Note that at the time of invocation of the above clause (transition rule) when run as a Prolog program, both Σ_1 *and* Σ_2 will be instantiated because of the way the clause is used by the rules for parallel composition later. This subtlety is not immediately obvious from the operational semantics, but in practice helps considerably with the implementation.

Of course, it is possible that the event does not occur, in which case there is no change of program state:

$$\frac{\langle \Sigma_1, \Sigma_2 \rangle \ \not\models \ \mathit{Event}}{@\mathit{Event} \cdot S \text{-}\langle \Sigma_1, \Sigma_2 \rangle\!\rightarrow @\mathit{Event} \cdot S.}$$

Alternatively, time may advance, with no change in the program state:

$$\frac{\mathsf{true}}{@\mathit{Event} \cdot S \text{-}\langle \_T \rangle\!\rightarrow @\mathit{Event} \cdot S.}$$

Note the prefix "_" used in the $\_T$ time *Prolog* variable (not to be confused with a Verilog variable) above. At this level we do not know (or even care) how many time units the program will advance. Thus $\_T$ could take on any value and is the only occurrence of this Prolog variable in the clause. The prefix "_" indicates to the Prolog system that we know this fact. Otherwise a warning message is issued by most Prolog systems when the program is loaded.

Verilog includes a fairly standard conditional "`if`" statement. If the Boolean condition is true, the first sub-program S_1 is executed. We do not care about the contents of the second sub-program $\_S_2$ in this case:

$$\frac{\mathit{EB} \cdot (\Sigma)}{\langle \texttt{if} \ (\mathit{EB}) \cdot S_1 \ \texttt{else} \ \_S_2, \Sigma \rangle \rightarrow \langle S_1, \Sigma \rangle.}$$

$\mathit{EB} \cdot (\Sigma)$ is true if the Boolean expression (condition) *EB* evaluates to true in the environment of the state Σ, holding the values of all the variables.

Alternatively the condition is false, in which case S_2 is executed instead:

$$\frac{\neg \ \mathit{EB} \cdot (\Sigma)}{\langle \texttt{if} \ (\mathit{EB}) \cdot \_S_1 \ \texttt{else} \ S_2, \Sigma \rangle \rightarrow \langle S_2, \Sigma \rangle.}$$

Note that the conditional construct does not introduce non-determinism since, although it is defined using two rules, the conditions under which the rules may be applied are disjoint. Thus only one of the rules is applicable at any particular time for a specific "`if`" statement. In addition, one of the rules *must* be applicable, so the condition on the two rules together is simply true.

Similarly, the iterative "`while`" loop has two cases to be considered. When the condition is true, the program within loop is executed once before the condition is considered again:

$$\frac{\mathit{EB} \cdot (\Sigma)}{\langle \texttt{while} \ (\mathit{EB}) \cdot S, \Sigma \rangle \rightarrow \langle S \ ; \ \texttt{while} \ (\mathit{EB}) \cdot S, \Sigma \rangle.}$$

When the condition is false, the program terminates immediately:

$$\frac{\neg \ \mathit{EB} \cdot (\Sigma)}{\langle \texttt{while} \ (\mathit{EB}) \cdot \_S, \Sigma \rangle \rightarrow \langle \epsilon, \Sigma \rangle.}$$

As with the `if` construct, one or other of these rules is always applicable, but never both together.

For sequential composition of two programs, if the first program terminates successfully, we can proceed to the second one immediately. The transition may allow time to pass or may involve events:

$$\frac{\langle S_1, \Sigma\rangle \rightarrow \langle \epsilon, \Sigma'\rangle}{\langle S_1 \ ; \ S_2, \Sigma\rangle \rightarrow \langle S_2, \Sigma'\rangle}.$$

$$\frac{S_1 \neq \epsilon \ \wedge \ S_1 \text{-}\langle T\rangle\!\rightarrow \epsilon}{S_1 \ ; \ S_2 \text{-}\langle T\rangle\!\rightarrow S_2}.$$

$$\frac{S_1 \text{-}\langle \Sigma_1, \Sigma_2\rangle\!\rightarrow \epsilon}{S_1 \ ; \ S_2 \text{-}\langle \Sigma_1, \Sigma_2\rangle\!\rightarrow S_2}.$$

We place the clauses above first in the list of Prolog clauses associated with sequential composition so that execution of the first sub-program S_1 to completion is considered first by the Prolog system if this is possible. Notice the check for S_1 not being ϵ when time advances. This is a subtle difference between ϵ and `skip` found in some other programming languages such as Occam [34]. An example will be shown later in Section 3.

Alternatively, S_1 may only partially progress:

$$\frac{\langle S_1, \Sigma\rangle \rightarrow \langle S_1', \Sigma'\rangle \ \wedge \ S_1' \neq \epsilon}{\langle S_1 \ ; \ S_2, \Sigma\rangle \rightarrow \langle S_1' \ ; \ S_2, \Sigma'\rangle}.$$

$$\frac{S_1 \text{-}\langle T\rangle\!\rightarrow S_1' \ \wedge \ S_1' \neq \epsilon}{S_1 \ ; \ S_2 \text{-}\langle T\rangle\!\rightarrow S_1' \ ; \ S_2}.$$

$$\frac{S_1 \text{-}\langle \Sigma_1, \Sigma_2\rangle\!\rightarrow S_1' \ \wedge \ S_1' \neq \epsilon}{S_1 \ ; \ S_2 \text{-}\langle \Sigma_1, \Sigma_2\rangle\!\rightarrow S_1' \ ; \ S_2}.$$

Note that the original operational semantics [18] and the syntax of Verilog itself both use the enclosing keywords `begin...end` for sequential composition blocks. This syntactic sugar adds unnecessary complication to the simulator. The Prolog parser will accept standard round brackets to disambiguate parsing of operators if required. The `begin...end` construct and other derived constructs are included separately as a macro definitions in the Prolog-based operational semantics for completeness [3].

This completes the presentation of the basic sequential features included in this semantics. The clauses above are very close to the original operational semantics on which this simulator is based [18].

2.2. Parallel composition

Next we consider parallel composition of sequential program components. We consider five cases, each including a base case and one or more inductive cases.

The first three sets of clauses deal respectively with entering, remaining in, and exiting sequential program executions not involving time delays or events. These are referred to as "*instantaneous sections*" in the rest of this paper.

To enter an instantaneous section, (at least) one of the parallel sequential programs must of capable of executing:

$$\frac{\langle S, \Sigma\rangle \rightarrow \langle S', \Sigma'\rangle}{\langle S, \Sigma, \langle\rangle, 0\rangle \longrightarrow \langle S', \Sigma', \Sigma, 1\rangle}.$$

$$\frac{\langle S_1, \Sigma, \langle\rangle, 0\rangle - \!\!\rightarrow \$_1', \Sigma', \Sigma, I\rangle \ \wedge \ I > 0}{\langle S_1 \ || \ S_2, \Sigma, \langle\rangle, 0\rangle - \!\!\rightarrow \$_1' \ || \ S_2, \Sigma', \Sigma, I\rangle}.$$

$$\frac{\langle S_2, \Sigma, \langle\rangle, 0\rangle - \!\!\rightarrow \$_2', \Sigma', \Sigma, I\rangle \ \wedge \ I > 0 \ \wedge \ J = I + 1}{\langle S_1 \ || \ S_2, \Sigma, \langle\rangle, 0\rangle - \!\!\rightarrow \$_1 \ || \ S_2', \Sigma', \Sigma, J\rangle}.$$

The label in the state (I and J above) is used as a numerical index to indicate which of the parallel sequential programs is currently executing, if any. The programs are simply numbered from left to right, from 1 upwards. A value of zero indicates that none of the parallel sequential programs are executing an instantaneous section. Of course, more than one of the sub-programs may be willing to enter an instantaneous section. The Prolog simulator searches all the possibilities in the order given in the overall Verilog program.

Once inside an instantaneous section, the Verilog program will continue to execute in a sequential manner if possible:

$$\frac{\langle S, \Sigma\rangle \rightarrow \langle S', \Sigma'\rangle}{\langle S, \Sigma, \Sigma_0, 1\rangle - \!\!\rightarrow \$', \Sigma', \Sigma_0, 1\rangle}.$$

$$\frac{I > 0 \ \wedge \ \langle S_1, \Sigma, \Sigma_0, I\rangle - \!\!\rightarrow \$_1', \Sigma', \Sigma_0, I\rangle}{\langle S_1 \ || \ S_2, \Sigma, \Sigma_0, I\rangle \longrightarrow \langle S_1' \ || \ S_2, \Sigma', \Sigma_0, I\rangle}.$$

$$\frac{I = J - 1 \ \wedge \ I > 0 \ \wedge \ \langle S_2, \Sigma, \Sigma_0, I\rangle \longrightarrow \langle S_2', \Sigma', \Sigma_0, I\rangle}{\langle S_1 \ || \ S_2, \Sigma, \Sigma_0, J\rangle \longrightarrow \langle S_1 \ || \ S_2', \Sigma', \Sigma_0, J\rangle}.$$

Otherwise the program will exit an instantaneous section that is currently being executed if it is impossible to progress any further:

$$\frac{\langle S, \Sigma\rangle \not\rightarrow}{\langle S, \Sigma, \Sigma_0, 1\rangle - \!\!\rightarrow \$, \Sigma, \Sigma_0, 0\rangle}.$$

$$\frac{I > 0 \ \wedge \ \langle S_1, \Sigma, \Sigma_0, I\rangle - \!\!\rightarrow \$_1', \Sigma, \Sigma_0, 0\rangle}{\langle S_1 \ || \ S_2, \Sigma, \Sigma_0, I\rangle \longrightarrow \langle S_1' \ || \ S_2, \Sigma, \Sigma_0, 0\rangle}.$$

$$\frac{I = J - 1 \ \wedge \ I > 0 \ \wedge \ \langle S_2, \Sigma, \Sigma_0, I\rangle \longrightarrow \langle S_2', \Sigma, \Sigma_0, 0\rangle}{\langle S_1 \ || \ S_2, \Sigma, \Sigma_0, J\rangle \longrightarrow \langle S_1 \ || \ S_2', \Sigma, \Sigma_0, 0\rangle}.$$

If no sequential program is engaged in an instantaneous section and all the parallel sequential programs are willing to engage in an event transition, this may occur:

$$\frac{S \multimap\langle \Sigma_0, \Sigma\rangle\rightarrowtail S'}{\langle S, \Sigma, \Sigma_0, 0\rangle \longrightarrow \langle S', \Sigma, \langle\rangle, 0\rangle.}$$

$$\frac{\langle S_1, \Sigma, \Sigma_0, 0\rangle \dashrightarrow \langle S_1', \Sigma, \langle\rangle, 0\rangle \ \wedge \ \langle S_2, \Sigma, \Sigma_0, 0\rangle \dashrightarrow \langle S_2', \Sigma, \langle\rangle, 0\rangle}{\langle S_1 \mid\mid S_2, \Sigma, \Sigma_0, 0\rangle \dashrightarrow \langle S_1' \mid\mid S_2', \Sigma, \langle\rangle, 0\rangle.}$$

Alternatively, if no sequential program is engaged in an instantaneous section and all the parallel sequential programs are willing to engage in allowing time to progress by T time units, then a timed transition can occur:

$$\frac{S \multimap\langle T\rangle\rightarrowtail S'}{\langle S, \Sigma, \langle\rangle, 0\rangle \multimap\langle T\rangle\longrightarrow \langle S', \Sigma, \langle\rangle, 0\rangle.}$$

$$\frac{\langle S_1, \Sigma, \langle\rangle, 0\rangle \multimap\langle T\rangle\longrightarrow \langle S_1', \Sigma, \langle\rangle, 0\rangle \ \wedge \ \langle S_2, \Sigma, \langle\rangle, 0\rangle \multimap\langle T\rangle\longrightarrow \langle S_2', \Sigma, \langle\rangle, 0\rangle}{\langle S_1 \mid\mid S_2, \Sigma, \langle\rangle, 0\rangle \multimap\langle T\rangle\longrightarrow \langle S_1' \mid\mid S_2', \Sigma, \langle\rangle, 0\rangle.}$$

The above clauses cover the parallel transitions given in the original operational semantics [18].

2.3. Additional transition rules

Some aspects in the original semantics [18] have been left informal, which is fine in mathematics, but not possible if the semantics is to be implemented as a simulator. For instance, once a program has executed to termination (ϵ), it can be eliminated as a parallel sequential program if it is the last one in the list. Earlier sub-programs should not be eliminated since this would affect the indexing of instantaneous sections. We could add such a rule as follows:

$$\frac{\langle S_2, \Sigma, \langle\rangle, 0\rangle \dashrightarrow \langle \epsilon, \Sigma, \langle\rangle, 0\rangle}{\langle S_1 \mid\mid S_2, \Sigma, \langle\rangle, 0\rangle \dashrightarrow \langle S_1, \Sigma, \langle\rangle, 0\rangle.}$$

With this rule, a set of parallel sub-programs will reduce to ϵ if it the overall parallel program terminates completely (i.e., all the component sequential programs terminate). However, such a rule can add considerably to the number of possible non-deterministic execution paths, so we omit it in the animations shown in Section 3. If required, we could ensure this rule is only used when no others are applicable (e.g., at the end of execution traces).

We must allow time to progress when sub-programs have finished:

$$\frac{\textsf{true}}{\epsilon \multimap\langle \_T\rangle\rightarrowtail \epsilon.}$$

For parallel triggering of guards, we must allow events to trigger if sequential sub-programs have terminated, are ready to perform a delay or to execute a sequential construct. For example:

$$\frac{\textsf{true}}{\epsilon \multimap\langle \_\Sigma_1, \_\Sigma_2\rangle\rightarrowtail \epsilon.}$$

Some rules are helpful in achieving notational simplicity:

$$\frac{S \multimap\langle T\rangle\rightarrowtail S'}{\langle S, \Sigma\rangle \multimap\langle T\rangle\rightarrowtail \langle S', \Sigma\rangle.}$$

$$\frac{S \multimap\langle \Sigma_1, \Sigma_2\rangle\rightarrowtail S'}{\langle S, \Sigma\rangle \multimap\langle \Sigma_1, \Sigma_2\rangle\rightarrowtail \langle S', \Sigma\rangle.}$$

2.4. Conditional transitions

So far, we have defined rules for various forms of transition rule, including conditions when those rules may be applied above the horizontal bar. The condition may simply be true in which case the rule may be applied unconditionally. It may be another rule, in which case, that rule must be checked. It may be some constraint; or it may be a combination of these conjoined together.

We can encode the various transition rules in the following form:

$$\frac{S \rightarrow S' \ \Leftarrow \ \mathit{Cond}}{S \rightarrow S', \{\mathit{Cond}\}.}$$

Here we use the standard *Constraint Logic Programming* (CLP) convention of enclosing constraint conditions within curly brackets "$\{\ldots\}$" [29]. Now we must encode all the conditions we need for this particular set of rules using further Prolog clauses. Fortunately this proves to be relatively simple. The clauses needed are in fact no larger than the original rules encoded above, and can be found in full elsewhere [3].

At the top level, transitions can be timed or untimed. We add the additional conditions that time advances or the program state changes to ensure progress:

$$\frac{S \multimap\langle T\rangle\rightarrowtail S' \ \Leftarrow \ \mathit{Cond}}{S \multimap\langle T\rangle\dashrightarrow S', \{\mathit{Cond} \ \wedge \ T > 0\}.}$$

$$\frac{S \multimap\langle 0\rangle\rightarrowtail S' \ \Leftarrow \ \mathit{Cond}}{S \dashrightarrow S', \{\mathit{Cond} \ \wedge \ S' \neq S\}.}$$

Now we can monitor traces of the parallel system state for terminating programs:

$$\mathit{trace}(\mathit{Sys}, \mathit{Sys} \multimap\langle T\rangle\rightarrowtail \mathit{Trace}) \ \Leftarrow \ \mathit{Sys} \multimap\langle T\rangle\rightarrowtail \mathit{Sys}', \mathit{trace}(\mathit{Sys}', \mathit{Trace}).$$
$$\mathit{trace}(\mathit{Sys}, \mathit{Sys}) \ \Leftarrow \ \neg(\mathit{Sys} \multimap\langle \_T\rangle\rightarrowtail \_\mathit{Sys}').$$

Checking for termination is done using Prolog's *negation as failure* [6], normally indicated using the "\+" operator in standard Prolog, but here indicated using the more logical "¬" operator. Prolog's negation as failure must be used with care. Essentially, if the (Prolog) variables are instantiated suitably at the time of invocation, such that the clause fails in a finite manner, this form of negation can be used safely.

It is convenient to execute a program with the variable and event states initialized to empty and with no individual parallel threads in an instantaneous section:

$$\mathit{run}(P, \mathit{Trace}) \quad \Leftarrow \quad \mathit{trace}(\langle P, \langle\rangle, \langle\rangle, 0\rangle, \mathit{Trace}).$$

This can be used to return successive possible finite execution traces to the user, for a particular instantiated program P.

3. Verilog Example Animations

In this section we now present a selection of example simulation runs of some simple Verilog programs using the simulator presented in the last section. In normal operation, Prolog will return possible answers to queries, in the order that they are found using its depth-first left to right search of the clauses. If an answer is found, the user has the option to request another answer or abort the search. If no answer can be found, the Prolog system simply responds with "No" and expects a fresh query from the user in response to its "?–" prompt.

The simulation includes additional support clauses to aid in the running of Verilog programs. Rather than continuously prompting the user, possible transitions are simply output to the display, using Prolog's built-in write(...) pseudo-clause. The simulator continues until all the possibilities are exhausted or a set number of time steps or transitions has been reached, in order to allow non-terminating programs to be aborted conveniently (using Prolog's built-in abort clause).

First, consider a single sequential program, consisting of the sequential composition of three assignments:

?– *run* $a = 1\ ;\ b = 2\ ;\ c = 3.$
$0 \quad \langle a = 1\ ;\ b = 2\ ;\ c = 3, \langle\rangle, \langle\rangle, 0\rangle$
$1 \longrightarrow \langle b = 2\ ;\ c = 3, \langle a = 1\rangle, \langle\rangle, 1\rangle$
$2 \longrightarrow \langle c = 3, \langle a = 1, b = 2\rangle, \langle\rangle, 1\rangle$
$3 \longrightarrow \langle \epsilon, \langle a = 1, b = 2, c = 3\rangle, \langle\rangle, 1\rangle$
$4 \longrightarrow \langle \epsilon, \langle a = 1, b = 2, c = 3\rangle, \langle\rangle, 0\rangle$
No

Here the overall "parallel" program (consisting of only a single sequential sub-program) must first enter the only instantaneous section available (thread 1). Then each assignment is executed, updating the variable state appropriately. Finally the program exits the instantaneous section when ϵ is reached. Once out of the instantaneous section there are no other possibilities; the parallel program has reached ϵ and thus the Prolog simulator returns "No" and expects another query from the user. No non-determinism is involved in this simple example.

Now consider a delay statement, allowing a delay of 3 time units:

?– *run* $\#3.$
$0 \quad \langle \#3, \langle\rangle, \langle\rangle, 0\rangle$
$1 \ \dashv\langle 3\rangle\mapsto \ \langle \epsilon, \langle\rangle, \langle\rangle, 0\rangle$
$1 \ \dashv\langle 2\rangle\mapsto \ \langle \#1, \langle\rangle, \langle\rangle, 0\rangle$
$2 \ \dashv\langle 1\rangle\mapsto \ \langle \epsilon, \langle\rangle, \langle\rangle, 0\rangle$
$1 \ \dashv\langle 1\rangle\mapsto \ \langle \#2, \langle\rangle, \langle\rangle, 0\rangle$
$2 \ \dashv\langle 2\rangle\mapsto \ \langle \epsilon, \langle\rangle, \langle\rangle, 0\rangle$
$2 \ \dashv\langle 1\rangle\mapsto \ \langle \#1, \langle\rangle, \langle\rangle, 0\rangle$
$3 \ \dashv\langle 1\rangle\mapsto \ \langle \epsilon, \langle\rangle, \langle\rangle, 0\rangle$
No

The most "preferable" execution is simply to allow time to progress by 3 time units and then finish executing the program. However there are three other possibilities. Either time could progress by 2 units followed by 1 unit, or it could progress by 1 unit followed by 2 units, or it could progress by 1 unit in three steps. Of course this non-determinism is not very interesting in the case of a single sequential program, but consider three parallel assignments, each delayed by a different amount:

?– *run* $\#3\ ;\ a = 1 \ ||\ \#1\ ;\ b = 2 \ ||\ \#2\ ;\ c = 3.$
$0 \quad \langle \#3\ ;\ a = 1 \ ||\ \#1\ ;\ b = 2 \ ||\ \#2\ ;\ c = 3, \langle\rangle, \langle\rangle, 0\rangle$
$1 \ \dashv\langle 1\rangle\mapsto \ \langle \#2\ ;\ a = 1 \ ||\ b = 2 \ ||\ \#1\ ;\ c = 3, \langle\rangle, \langle\rangle, 0\rangle$
$2 \ \longrightarrow \ \langle \#2\ ;\ a = 1 \ ||\ \epsilon \ ||\ \#1\ ;\ c = 3, \langle b = 2\rangle, \langle\rangle, 2\rangle$
$3 \ \longrightarrow \ \langle \#2\ ;\ a = 1 \ ||\ \epsilon \ ||\ \#1\ ;\ c = 3, \langle b = 2\rangle, \langle\rangle, 0\rangle$
$4 \ \dashv\langle 1\rangle\mapsto \ \langle \#1\ ;\ a = 1 \ ||\ \epsilon \ ||\ c = 3, \langle b = 2\rangle, \langle\rangle, 0\rangle$
$5 \ \longrightarrow \ \langle \#1\ ;\ a = 1 \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3\rangle, \langle b = 2\rangle, 3\rangle$
$6 \ \longrightarrow \ \langle \#1\ ;\ a = 1 \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3\rangle, \langle b = 2\rangle, 0\rangle$
$7 \ \longrightarrow \ \langle \#1\ ;\ a = 1 \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3\rangle, \langle\rangle, 0\rangle$
$8 \ \dashv\langle 1\rangle\mapsto \ \langle a = 1 \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3\rangle, \langle\rangle, 0\rangle$
$9 \ \longrightarrow \ \langle \epsilon \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3, a = 1\rangle, \langle b = 2, c = 3\rangle, 1\rangle$
$10 \ \longrightarrow \ \langle \epsilon \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3, a = 1\rangle, \langle b = 2, c = 3\rangle, 0\rangle$
$11 \ \longrightarrow \ \langle \epsilon \ ||\ \epsilon \ ||\ \epsilon, \langle b = 2, c = 3, a = 1\rangle, \langle\rangle, 0\rangle$
No

The simulator attempts to progress time by the maximal possible amount and executes the assignments after the required delays. The non-determinism in the delay construct rule allows this to be done in a natural way, letting the Prolog system explore the possible paths for us, and resolving the non-determinism in the process. In this particular case there is only one possible execution path since the difference between the delays is only 1 time unit, the minimum allowable time delay.

Of course, parallelism may introduce non-determinism. If two (or more) processes are willing to engage in an instantaneous section, then they may be executed in any order:

?– *run* $a = 1 \mid\mid b = 2$.
0 $\langle a = 1 \mid\mid b = 2, \langle\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle \epsilon \mid\mid b = 2, \langle a = 1\rangle, \langle\rangle, 1\rangle$
2 $-\!\!\rightarrow \langle \epsilon \mid\mid b = 2, \langle a = 1\rangle, \langle\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle a = 1, b = 2\rangle, \langle a = 1\rangle, 2\rangle$
4 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle a = 1, b = 2\rangle, \langle a = 1\rangle, 0\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle a = 1, b = 2\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle a = 1 \mid\mid \epsilon, \langle b = 2\rangle, \langle\rangle, 2\rangle$
2 $-\!\!\rightarrow \langle a = 1 \mid\mid \epsilon, \langle b = 2\rangle, \langle\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle b = 2, a = 1\rangle, \langle b = 2\rangle, 1\rangle$
4 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle b = 2, a = 1\rangle, \langle b = 2\rangle, 0\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle b = 2, a = 1\rangle, \langle\rangle, 0\rangle$
No

With two parallel instantaneous sections, there are two possibilities; either the first section is executed completely followed by the second section, or vice versa. Thus if the same variable is assigned different values in different parallel instantaneous sections, it is non-deterministic which value it will hold after the sections have been executed:

?– *run* $v = 1 \mid\mid v = 2$.
0 $\langle v = 1 \mid\mid v = 2, \langle\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle \epsilon \mid\mid v = 2, \langle v = 1\rangle, \langle\rangle, 1\rangle$
2 $-\!\!\rightarrow \langle \epsilon \mid\mid v = 2, \langle v = 1\rangle, \langle\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 2\rangle, \langle v = 1\rangle, 2\rangle$
4 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 2\rangle, \langle v = 1\rangle, 0\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 2\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle v = 1 \mid\mid \epsilon, \langle v = 2\rangle, \langle\rangle, 2\rangle$
2 $-\!\!\rightarrow \langle v = 1 \mid\mid \epsilon, \langle v = 2\rangle, \langle\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 2\rangle, 1\rangle$
4 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 2\rangle, 0\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1\rangle, \langle\rangle, 0\rangle$
No

Now consider an event guard in parallel with a delayed assignment than triggers that event:

?– *run* $@v{\cdot}w = 1 \mid\mid v = 0\ ;\ \#1\ ;\ v = 1$.
0 $\langle @v{\cdot}w = 1 \mid\mid v = 0\ ;\ \#1\ ;\ v = 1, \langle\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \#1\ ;\ v = 1, \langle v = 0\rangle, \langle\rangle, 2\rangle$
2 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \#1\ ;\ v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$
3 $-\!\langle 1\rangle\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$
4 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 2\rangle$
5 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 0\rangle$
6 $-\!\!\rightarrow \langle w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle\rangle, 0\rangle$
7 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 1\rangle$
8 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 0\rangle$
9 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle\rangle, 0\rangle$
No

The delay must occur first since the event cannot occur until after the delay. The assignment $v = 1$ changes the value of the variable v from 0 to 1, thus causing the event $@v$ to occur. This allows the guarded assignment $w = 1$ to be executed and the program to terminate.

A rather more subtle case can occur with the occurrence of ϵ at the start of a sequential composition. Consider the case with an event guard. In the following example, a trace where v is initially known to be 0, there are two possibilities that can occur non-deterministically:

?– *trace* $\langle \epsilon\ ;\ @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$.
0 $\langle \epsilon\ ;\ @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle \epsilon\ ;\ @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 2\rangle$
2 $-\!\!\rightarrow \langle \epsilon\ ;\ @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$
2 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 2\rangle$
3 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 0\rangle$
4 $-\!\!\rightarrow \langle w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle\rangle, 0\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 1\rangle$
6 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 0\rangle$
7 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle\rangle, 0\rangle$
No

Note that one execution path blocks expecting an event $@v$ that can never occur, whereas the other path proceeds to successful completion. However, without the leading ϵ in the first thread of the parallel program, only one of these execution paths is possible:

?– *trace* $\langle @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$.
0 $\langle @v{\cdot}w = 1 \mid\mid v = 1, \langle v = 0\rangle, \langle\rangle, 0\rangle$
1 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 2\rangle$
2 $-\!\!\rightarrow \langle @v{\cdot}w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle v = 0\rangle, 0\rangle$
3 $-\!\!\rightarrow \langle w = 1 \mid\mid \epsilon, \langle v = 1\rangle, \langle\rangle, 0\rangle$
4 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 1\rangle$
5 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle v = 1\rangle, 0\rangle$
6 $-\!\!\rightarrow \langle \epsilon \mid\mid \epsilon, \langle v = 1, w = 1\rangle, \langle\rangle, 0\rangle$
No

Thus ϵ is rather different from `skip`, as found in Occam for example [34]. The simulator has proved useful for checking the semantics in such situations.

The original operational semantics includes an example trace for a program, hand "executed" by the authors [18]. Figure 1 shows the result of running that program (with the arbitrary program S in the original example replaced by $\#1$ so that the program is fully instantiated for the Prolog simulator). Eventually all the other sub-programs apart from the `while` loop terminate, triggering events on the variable v twice. After executing the delay, the loop then waits for another event that never occurs. In practice it would simply wait forever, but the trace here cannot continue and so terminates. It would be possible to change the simulator to continuously progress time at this point if this is desired instead. Note that the Prolog simulator can easily allow the number of time steps (or transition steps) until abortion to be set to any desired value if required.

The animation of such programs allows the semantics of Verilog to be explored in a simple manner, with the extra confidence that machine execution of a program brings compared to hand execution. Once formulated, it is easy to

$$
\begin{array}{lll}
?- & \textit{trace} & \langle \texttt{forever}\ @v\cdot(@v\cdot\#1) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle\rangle, 0\rangle. \\
0 & & \langle \texttt{forever}\ @v\cdot(@v\cdot\#1) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle\rangle, 0\rangle \\
1 & -\!\!\rightarrow & \langle \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle v=0\rangle, 1\rangle \\
2 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle v=0\rangle, 1\rangle \\
3 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle v=0\rangle, 0\rangle \\
4 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel v = v+1\ ;\ v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=0\rangle, \langle\rangle, 0\rangle \\
5 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel v = v+2 \parallel \#3\ ;\ v = v+3, \langle v=1\rangle, \langle v=0\rangle, 2\rangle \\
6 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \#3\ ;\ v = v+3, \langle v=3\rangle, \langle v=0\rangle, 2\rangle \\
7 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \#3\ ;\ v = v+3, \langle v=3\rangle, \langle v=0\rangle, 0\rangle \\
8 & -\!\!\rightarrow & \langle @v\cdot\#1\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \#3\ ;\ v = v+3, \langle v=3\rangle, \langle\rangle, 0\rangle \\
9 & -\langle 3\rangle\!\rightarrow & \langle @v\cdot\#1\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel v = v+3, \langle v=3\rangle, \langle\rangle, 0\rangle \\
10 & -\!\!\rightarrow & \langle @v\cdot\#1\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle v=3\rangle, 3\rangle \\
11 & -\!\!\rightarrow & \langle @v\cdot\#1\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle v=3\rangle, 0\rangle \\
12 & -\!\!\rightarrow & \langle \#1\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle\rangle, 0\rangle \\
13 & -\langle 1\rangle\!\rightarrow & \langle \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle\rangle, 0\rangle \\
14 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle v=6\rangle, 1\rangle \\
15 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle v=6\rangle, 0\rangle \\
16 & -\!\!\rightarrow & \langle @v\cdot(@v\cdot\#1)\ ;\ \texttt{while}\ \textsf{true}\cdot(@v\cdot(@v\cdot\#1)) \parallel \epsilon \parallel \epsilon, \langle v=6\rangle, \langle\rangle, 0\rangle
\end{array}
$$

Figure 1. Example trace, based on the example in [18].

add and remove rules to and from the Prolog simulator, and to modify them, to explore the consequences.

Of course, the simulator presented here does not cover the whole of Verilog. However, if further program constructs are considered important, their operational semantics transition rules can be formulated and added to the simulator relatively easily.

This exercise is considered a success in that a mathematical operational semantics for a significant subset of Verilog has been very directly encoded in Prolog in a short period of time. Of course this rapid-prototype system is only suitable for very small Verilog programs, but it could be a useful aid in the understanding of the semantics of Verilog programs which is otherwise only generally available in large informal documents such as the IEEE Standard [24] and textbooks [11] or in large software simulators that are necessarily deterministic for efficiency reasons.

The execution time for all the simple examples shown is essentially instantaneous, so the simulator is very usable for such didactic examples in practice. Of course the full search space is potentially exponential in the presence of non-determinism, so returning all possible execution paths for larger examples could be unacceptably slow and the amount of information returned daunting. However, with careful ordering of the clauses, the more interesting of the possible execution paths can be returned first. If just a single execution path is required, the simulator could be made acceptably efficient for much larger examples.

4. Conclusion

The Prolog simulator operational semantics for a subset of the Verilog Hardware Description Language (HDL) presented in this paper is pleasingly close to the original operation semantics on which it was based [18]. Much of the development time was spent on establishing the correct precedence and associativity of the operators and then ensuring the ordering of the encoded relations resulted in a preferred possible execution path being presented to the user first. An advantage of the logic programming approach, over the functional programming approach for instance, is that different execution paths can be allowed in a natural and implicit manner. This is especially useful in the execution of parallel programs (as normally required in the simulation of physically parallel hardware) since the combinations of potential execution paths can be explored in a convenient way by the Prolog system, with little or no explicit encoding for non-deterministic aspects required.

An interesting area for further exploration is the algebraic laws associated with Verilog [19]. The parallel aspects of Verilog mean that some laws associated with traditional sequential languages only apply in certain circumstances and additional laws are required for the parallel parts of the language. Using these laws, a compiler (written in Prolog, for example) from high-level Verilog programs to low-level digital hardware circuits could be produced.

Ultimately, formalization of hardware/software co-design to help achieve a unified framework for computer system development is a goal worth pursuing [15, 16, 17, 22]. It is possible to use a parallel programming language such as Occam in a unified manner for this purpose [25]. However, although attractive formally, unfortunately it is unlikely to be widely used in practice since Occam is not popular in industry. The choice of language is important, and a C-like co-design language (such as Handel-C [32]) may prove to be accepted more easily. A combination of an existing widely used HDL such as Verilog (or a subset of it) and a more traditional popular programming language (such as Java, for example) may be appropriate, provided the formal basis for each can be established and integrated.

References

[1] Bawankule, R., *Alternative Verilog FAQ*, 1997–2000.
URL: *http://www.angelfire.com/in/verilogfaq/*

[2] Bertot, Y. and Fraer, R., Reasoning with Executable Specifications. In P.D. Mosses, M. Nielsen and M.I. Schwartzbach (eds.), *TAPSOFT '95: Theory and Practice of Software Development*, Springer-Verlag, Berlin, Lecture Notes in Computer Science (LNCS) 915, pages 531–545, 1995,

[3] Bowen, J.P., *Animating the Semantics of VERILOG using Prolog*. UNU/IIST Technical Report no. 176, International Institute for Software Technology, United Nations University, Macau, September 1999.
URL: *ftp://ftp.iist.unu.edu/pub/techreports/report176.ps.gz*

[4] Bowen, J.P. and Hinchey, M.G., Formal Models and the Specification Process. In A.B. Tucker, Jr. (ed.), *The Computer Science and Engineering Handbook*, CRC Press, Section X, Software Engineering, Chapter 107, pages 2302–2322, 1997.

[5] Bowen, J.P. and Hinchey, M.G., *High-Integrity System Specification and Design*. Springer-Verlag, London, Formal Approaches to Computing and Information Technology (FACIT) series, 1999. URL: *http://www.fmse.cs.reading.ac.uk/hissd/*

[6] Clark, K.L., Negation as Failure. In H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plemum Press, New York, pages 293–322, 1978.

[7] Clocksin, W.F., Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, 4(1):59–82, March 1987.

[8] Clocksin, W.F. and Mellish C.S., *Programming in PROLOG*. Springer-Verlag, Berlin, 1994.

[9] Delgado Kloos, C. and Breuer, P.T. (eds.), *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[10] Fuchs, N.E., Specifications are (Preferably) Executable. In Bowen, J.P. and Hinchey, M.G. [5], pages 583–607, 1999.

[11] Golze, U., *VLSI Chip Design with the Hardware Description Language VERILOG*. Springer-Verlag, Berlin, 1996.

[12] Gordon, M., The Semantic Challenge of Verilog HDL. In *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, San Diego, California, USA, 26–29 June 1995. IEEE Computer Society Press, 1995.

[13] Gordon, M., *Event and Cycle Semantics of Hardware Description Languages*, University of Cambridge Computer Laboratory, UK, 16 January 1998. Version 1.4.

[14] Hayes, I.J. and Jones, C.B., Specifications are not (Necessarily) Executable. In J.P. Bowen and M.G. Hinchey [5], pages 563–581, 1999.

[15] He Jifeng, *A Behavioural Model for Co-design*. UNU/IIST Report No. 16, International Institute for Software Technology, United Nations University, Macau, May 1999.

[16] He Jifeng, A Common Framework for Mixed Hardware/Software Systems. In Araki, K., Galloway, A. and Taguchi, K. (eds.), *IFM 99: Proceedings of the 1st International Conference on Integrated Formal Methods*, York, 28–29 June 1999. Springer-Verlag, Berlin, pages 1–15, 1999.
URL: *http://www.cs.york.ac.uk/ifm-99/*

[17] He Jifeng, An Integrated Approach to Hardware/Software Co-design. In *Proc. IFIP WCC2000*, Beijing, China, 2000.

[18] He Jifeng and Xu Qiwen, An Operational Semantics of a Simulator Algorithm. International Institute for Software Technology, United Nations University, Macau, 1999. Revised version in *Proc. PDPTA2000*, Las Vegas, USA, 2000.

[19] He Jifeng and Xu Qiwen, *Verilog Programming*. International Institute for Software Technology, United Nations University, Macau, August 1999.

[20] Hinchey, M.G. and Bowen, J.P. (eds.), *Applications of Formal Methods*. Prentice Hall International Series in Computer Science, 1995.

[21] Hinchey, M.G. and Bowen, J.P. (eds.), *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, London, Formal Approaches to Computing and Information Technology (FACIT) series, 1999. URL: *http://www.fmse.cs.reading.ac.uk/isfm/*

[22] Hoare, C.A.R. and He Jifeng, *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.

[23] *IEEE Standard VHDL Language Reference Manual*. IEEE Standard 1076-1993, 1993.
URL: *http://standards.ieee.org/catalog/design.html#1076-1993*
See also: *IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS Changes)*. IEEE Standard 1076.1-1999, Approved Draft, 1999.

[24] *IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language*. IEEE Standard 1364-1995, 1995.
URL: *http://standards.ieee.org/catalog/design.html#1364-1995*

[25] Iyoda, J., Sampaio A. and Silva, L., ParTS: A Partitioning Transformation System. In J.M. Wing, J. Woodcock and J. Davis (eds.), *FM'99 – Formal Methods*. Springer-Verlag, Berlin, Lecture Notes in Computer Science (LNCS) 1709, pages 1400-1419, 1999.

[26] Knuth, D., Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[27] Knuth, D., *Literate Programming*. Center for the Study of Language and Information, Stanford, California, USA, CSLI Lecture Notes, no. 27, 1992. 4th printing, 1998.
URL: *http://www-cs-faculty.stanford.edu/~knuth/lp.html*

[28] Lamport, L., *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, 1994.

[29] Marriott, K. and Stuckey, P.J., *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[30] Nielson, H.R. and Nielson, F., *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992. (*Operational Semantics*, Chapter 2, pages 19–61.) Revised and available on-line, 1999.
URL: *http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html*

[31] Open Verilog International, *OVI_WEB*, 2000.
URL: *http://www.ovi.org/*

[32] Page, I., Constructing Hardware/Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[33] Plotkin, G., *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981. Reprinted 1991.

[34] Roscoe, A.W. and Hoare, C.A.R., The Laws of Occam Programming. *Theoretical Computer Science*, 60:177–229, 1988.

[35] Sagdeo, V., Pixley, C., *et al.*, *OVI Formal Verification Standards Working Group*, Open Verilog International, 1999.
URL: *http://www.eda.org/vfv/*

[36] Schneider, G. and Xu Qiwen, *Towards an Operational Semantics of Verilog*. UNU/IIST Report No. 147, International Institute for Software Technology, United Nations University, Macau, October 1998.

[37] Van Dung, T. and He Jifeng, *A Theory of Combinational Programs*. UNU/IIST Report No. 162, International Institute for Software Technology, United Nations University, Macau, April 1999.

[38] *VHDL International [VI]*, 1995–1999.
URL: *http://www.vhdl.org/*