The United Nations University

UNU/IIST

International Institute for
Software Technology

Towards an Operational Semantics of Verilog

Gerardo Schneider and Xu Qiwen

October 1998

UNU/IIST Report No. 147

$\boxed{\mathcal{R}}$

UNU/IIST and UNU/IIST Reports

UNU/IIST is a Research and Training Center of the United Nations University. It was founded in 1992, and is located in Macau. UNU/IIST is jointly funded by the Governor of Macau and the Governments of China and Portugal through contribution to the UNU Endowment Fund.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. advanced development projects in which software techniques supported by tools are applied,
2. research projects in which new techniques for software development are investigated,
3. curriculum development projects in which courses of software technology for universities in developing countries are developed,
4. courses which typically teach advanced software development techniques,
5. events in which conferences and workshops are organised or supported by UNU/IIST, and
6. dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research $\boxed{\text{R}}$, Technical $\boxed{\text{T}}$, Compendia $\boxed{\text{C}}$ or Administrative $\boxed{\text{A}}$. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Towards an Operational Semantics of Verilog

Gerardo Schneider and Xu Qiwen

Abstract

We give an operational semantics of simple version of Verilog hardware description language.

Gerardo Schneider is a fellow of UNU/IIST, on leave from Catholic University of Pelotas, Brazil, where he is a lecturer.

Xu Qiwen is a Research Fellow of UNU/IIST. His research interest is in Formal Techniques of Programming, including Theory for Concurrency and Real Time, Verification and Design Calculi. E-mail: qxu@iist.unu.edu

Contents

1 Introduction **1**

2 A Subset of Verilog and its Operational Semantics **1**

2.1 Procedural Assignment 2
2.2 Delay 2
2.3 Wait 3
2.4 Event control 4
2.5 Continuous Assignment 5
2.6 Sequential Composition 5
2.7 Conditional 6
2.8 While 6
2.9 Parallelism 7

3 Discussion **9**

1 Introduction

Modern hardware design typically uses hardware description languages to express designs at various levels of abstraction. A hardware description language is a high level programming language, with the usual programming constructs such as assignments, conditionals and iterations, and appropriate extensions for real-time, concurrency and data structures suitable for modelling hardware. The common approach is to first build a high level design using programming constructs. The high level design is then recoded using a subset of the description language which is closer to implementation. This process may be repeated several times until the design is at a sufficiently lower level such that the hardware can be synthesised from it.

For ensuring correctness of the development, a precise understanding of the description language used is apparently. Verilog is a hardware description language widely used in industry, but its standard semantics [5] is informal. A formal semantics will also be the basis of further formal support for the language. This includes methods to prove that the highest level description satisfies the overall requirements and that a lower level description correctly implements a higher level one.

Verilog programs can exhibit a rich variety of computations, when several features of the language are intertwined. The features include

- shared states, updated by possibly instantaneous assignments from different processes;
- delay statements;
- synchronisation by waiting for some conditions to become true;
- recursion.

It is therefore a non-trivial task to give an adequate semantics to the language. In contrast to many attempts to formalise semantics of VHDL, another popular but reportedly less used hardware description language, there is only a little work on formal semantics of Verilog [3, 6]. In this paper, we give a formal semantics to a large subset of Verilog using the operational approach of Plotkin [7].

2 A Subset of Verilog and its Operational Semantics

Gordon [3] suggested a simple version of Verilog which he called V. In this paper, for simplicity, we consider a subset of V which that has essentially the same statements as V except the function definition, the non-blocking assignment, the `disable` statement and the assignment statements with delays. The language is simple enough for experiments in formalisation, but contains sufficient features for being practically relevant.

The standard way to give an operational semantics is to define a set of transition rules over configurations, which are pairs of programs and states, such that any computation of a program can be generated from the transition rules. Due to the diversity of Verilog statements, we divide the transitions into three groups. More precisely, the three kinds of transitions are denoted as

$$\langle S, \sigma\rangle \xrightarrow{A} \langle S', \sigma'\rangle$$
$$\langle S, \sigma\rangle \xrightarrow{P} \langle S', \sigma'\rangle$$
$$\langle S, \sigma\rangle \xrightarrow{t} \langle S, \sigma\rangle$$

where S and S' are programs, σ and σ' are states (valuations of program variables). A transition represents executing the program by one step, and the program that remains to be executed is S'. The three transitions denote respectively an autonomous, a passive and a time advance steps. An autonomous step is one that can be activated by S itself, whereas a passive step must be initiated by an autonomous step of another process. Both autonomous and passive transitions are assumed to be instantaneous, often called discrete. In the time advance transition, t is a positive real representing the time elapsed, while program variables are not changed.

The general view is that the execution of a Verilog program consists of interleaving discrete transitions from concurrent processes. If there are discrete transitions enabled, one of them will be selected for execution, followed possibly by some passive steps. When none of the discrete transitions are enabled, time may advance until one discrete transition is enabled. We next give the transition rules for each statement. For brevity, we shall not repeat semantics of similar statements.

2.1 Procedural Assignment

The execution of procedural assignment $v = e$ is assumed to be atomic, with variable v assigned the value of expression e and other variables unchanged. The assignment is autonomous.

$$\langle v = e, \sigma\rangle \xrightarrow{A} \langle \epsilon, \sigma(v \mapsto e(\sigma))\rangle$$

where ϵ denotes the empty program, and $\sigma(v \mapsto e(\sigma))$ is same as σ except mapping v to $e(\sigma)$.

Example. $\langle v = v + 1, \sigma[v = 0]\rangle \xrightarrow{A} \langle \epsilon, \sigma[v = 1]\rangle$. By $\sigma[v = a]$, we denote that the value of v is a in the state.

2.2 Delay

The delay statement in Verilog is denoted by $\#n$. Its meaning is that when this instruction is reached, the process is blocked for n time units. During this period, other processes may execute discrete state transitions. Therefore, it is necessary to include transitions which advance time by less than n units, so that intermediate configurations. Discrete transitions from other processes can then take place from the intermediate configurations.

1. $$\langle \#n, \sigma \rangle \xrightarrow{n} \langle \epsilon, \sigma \rangle$$

2. $$\frac{n' < n}{\langle \#n, \sigma \rangle \xrightarrow{n'} \langle \#(n - n'), \sigma \rangle}$$

where the transition labels $t = n$ and $t = n'$ indicate respectively that n and n' units of time have passed.

Example.

$$\langle \#2, \sigma \rangle \xrightarrow{2} \langle \epsilon, \sigma \rangle$$
$$\langle \#2, \sigma \rangle \xrightarrow{1} \langle \#1, \sigma \rangle$$

2.3 Wait

The wait statement `wait` (eb) S evaluates Boolean expression eb, and if the value is true, then the control passes to the following statement S, otherwise, the wait statement is blocked until the condition becomes true.

1. $$\frac{eb(\sigma)}{\langle \texttt{wait}\ \ (eb)\ S, \sigma \rangle \xrightarrow{A} \langle S, \sigma \rangle}$$

2. $$\frac{\neg eb(\sigma)}{\langle \texttt{wait}\ \ (eb)\ S, \sigma \rangle \xrightarrow{t} \langle \texttt{wait}\ \ (eb)\ S, \sigma \rangle}$$

where $t \in \texttt{real}^+$ and $\texttt{real}^+$ is the set of positive reals.

Example.

$$\langle \texttt{wait}\ \ (v > 0)\ S, \sigma[v = 1] \rangle \xrightarrow{A} \langle S, \sigma[v = 1] \rangle$$
$$\langle \texttt{wait}\ \ (v < 0)\ S, \sigma[v = 1] \rangle \xrightarrow{2} \langle \texttt{wait}\ \ (v < 0)\ S, \sigma[v = 1] \rangle$$

2.4 Event control

Synchronisation can also be achieved by waiting for some events (state changes) to occur. The statement is of the form

$@(event)\ S;$

where *event* is considered as a binary state predicate, and the control is passed to S when *event* is satisfied. An event expression is of the following form

- v, indicating waiting for a change of the value in v,
- `posedge` v, indicating waiting for a positive change of the value in v,
- `negedge` v, indicating waiting for a negative change of the value in v,
- $(event_1 \texttt{or} \ \ldots \texttt{or} \ event_n)$, indicating waiting for any event to be true.

Given a pair of states σ and σ', an event expression is evaluated as follows

$$
\begin{array}{l}
@(v)(\sigma, \sigma') \stackrel{\text{def}}{=} (\sigma(v) \neq \sigma'(v)) \\
@(\texttt{posedge}\ v)(\sigma, \sigma') \stackrel{\text{def}}{=} (\sigma(v) < \sigma'(v)) \\
@(\texttt{negedge}\ v)(\sigma, \sigma') \stackrel{\text{def}}{=} (\sigma(v) > \sigma'(v)) \\
@(event_1 \texttt{or}\ \ldots \texttt{or}\ event_n) \stackrel{\text{def}}{=} (@(event_1)(\sigma, \sigma') \vee \ldots \vee @(event_n)(\sigma, \sigma'))
\end{array}
$$

Having defined the interpretation of event expressions, it is easy to give the transition rules

1. $$\frac{@(event)(\sigma, \sigma')}{\langle @(event)\ S, \sigma \rangle \xrightarrow{P} \langle S, \sigma' \rangle}$$

2. $$\frac{\neg @(event)(\sigma, \sigma')}{\langle @(event)\ S, \sigma \rangle \xrightarrow{t} \langle @(event)\ S, \sigma' \rangle}$$

Note that the first transition is passive, that is, the state change must be caused by another process.

Example.

$$\begin{array}{l}
\langle @(v)\ S, \sigma[v=0]\rangle \xrightarrow{P} \langle S, \sigma'[v=1]\rangle \\
\langle @(v)\ S, \sigma[v=0]\rangle \xrightarrow{2} \langle S, \sigma[v=0]\rangle \\
\langle @(\texttt{posedge}\ \ v)\ S, \sigma[v=0]\rangle \xrightarrow{P} \langle S, \sigma'[v=1]\rangle \\
\langle @(\texttt{negedge}\ \ v)\ S, \sigma[v=0]\rangle \xrightarrow{2} \langle @(\texttt{negedge}\ \ v)\ S, \sigma[v=0]\rangle \\
\langle @(v\ \texttt{or}\ \ \texttt{posedge}\ \ v)\ S, \sigma[v=0]\rangle \xrightarrow{P} \langle S, \sigma'[v=1]\rangle \\
\langle @(v\ \texttt{or}\ \ \texttt{negedge}\ \ v)\ S, \sigma[v=0]\rangle \xrightarrow{2} \langle @(v\ \texttt{or}\ \ \texttt{negedge}\ \ v)\ S, \sigma[v=0]\rangle
\end{array}$$

2.5 Continuous Assignment

A continuous assignment `assign` $w = e$ models input, denoted by expression e, to a wire w. The statement continuously watches for changes to variables in its righthand side (variables that occur on the expression e) and whenever a change happens expression e is re-evaluated and the result is immediately propagated to w. Time advances when no variables are changed.

1. $$\frac{e(\sigma) \neq e(\sigma')}{\langle \texttt{assign}\ \ w = e, \sigma\rangle \xrightarrow{P} \langle \texttt{assign}\ \ w = e, \sigma'(w \mapsto e(\sigma'))\rangle}$$

2. $$\langle \texttt{assign}\ \ w = e, \sigma\rangle \xrightarrow{t} \langle \texttt{assign}\ \ w = e, \sigma\rangle$$

As in the event control statements, the state transition is passive.

Example.

$$\begin{array}{l}
\langle \texttt{assign}\ \ w = v+1, \sigma[v = 0 \wedge w = 1]\rangle \xrightarrow{P} \langle \texttt{assign}\ \ w = v+1, \sigma'[v = 1 \wedge w = 2]\rangle \\
\langle \texttt{assign}\ \ w = v+1, \sigma[v = 0]\rangle \xrightarrow{2} \langle \texttt{assign}\ \ w = v+1, \sigma[v = 0]\rangle
\end{array}$$

2.6 Sequential Composition

A sequential composition is of the form `begin` $S_1; \ldots; S_n$ `end` and its execution is that of the S_1 followed by the execution of `begin` $S_2; \ldots; S_n$ `end`.

1. $$\frac{\langle S_1, \sigma\rangle \xrightarrow{\delta} \langle S_1', \sigma'\rangle \wedge S_1' \neq \epsilon}{\langle \texttt{begin}\ \ S_1; S_2; \ldots; S_n\ \ \texttt{end}, \sigma\rangle \xrightarrow{\delta} \langle \texttt{begin}\ \ S_1'; S_2; \ldots; S_n\ \ \texttt{end}, \sigma'\rangle}$$

2. $$\frac{\langle S_1, \sigma\rangle \xrightarrow{\delta} \langle \epsilon, \sigma'\rangle}{\langle \texttt{begin}\ \ S_1; S_2; \ldots; S_n\ \ \texttt{end}, \sigma\rangle \xrightarrow{\delta} \langle \texttt{begin}\ \ S_2; \ldots; S_n\ \ \texttt{end}, \sigma'\rangle}$$

where $\delta \in (\{A, P\} \cup \texttt{real}^+)$.

Example.

$$\langle \texttt{begin}\ \ v = v+1; S\ \ \texttt{end}, \sigma[v=0]\rangle \xrightarrow{A} \langle S, \sigma[v=1]\rangle$$
$$\langle \texttt{begin}\ \ \#2; S\ \ \texttt{end}, \sigma\rangle \xrightarrow{1} \langle \texttt{begin}\ \ \#1; S\ \ \texttt{end}, \sigma\rangle$$
$$\langle \texttt{begin assign}\ \ w = v+1; S\ \ \texttt{end}, \sigma[v=0 \wedge w=1]\rangle$$
$$\xrightarrow{P} \langle \texttt{begin assign}\ \ w = v+1; S\ \ \texttt{end}, \sigma'[v=1 \wedge w=2]\rangle$$

2.7 Conditional

The conditional statement `if` (eb) S_1 `else` S_2 has the usual meaning: if the expression eb evaluates to true, then S_1 is executed, otherwise S_2 is executed. The selection is autonomous.

1. $$\frac{eb(\sigma)}{\langle \texttt{if}\ \ (eb)\ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \xrightarrow{A} \langle S_1, \sigma\rangle}$$

2. $$\frac{\neg eb(\sigma)}{\langle \texttt{if}\ \ (eb)\ S_1\ \ \texttt{else}\ \ S_2, \sigma\rangle \xrightarrow{A} \langle S_2, \sigma\rangle}$$

Example.

$$\langle \texttt{if}\ \ (v > -1)\ S_1\ \ \texttt{else}\ \ S_2, \sigma[v=0]\rangle \xrightarrow{A} \langle S_1, \sigma[v=0]\rangle$$
$$\langle \texttt{if}\ \ (v > 1)\ S_1\ \ \texttt{else}\ \ S_2, \sigma[v=0]\rangle \xrightarrow{A} \langle S_2, \sigma[v=0]\rangle$$

2.8 While

The iteration statement `while` (eb) S is interpreted as usual: if eb is evaluated to true, then S is executed and followed by the original iteration statement, and if eb is evaluated to false, then the iteration statement terminates.

1. $$\frac{eb(\sigma)}{\langle \texttt{while}\ \ (eb)\ S, \sigma\rangle \xrightarrow{A} \langle S; \texttt{while}\ \ (eb)\ S, \sigma\rangle}$$

2. $$\frac{\neg eb(\sigma)}{\langle \texttt{while}\ (eb)\ S, \sigma\rangle \xrightarrow{A} \langle \epsilon, \sigma\rangle}$$

Example.

$$\langle \texttt{while}\ (v > -1)\ S, \sigma[v=0]\rangle \xrightarrow{A} \langle S; \texttt{while}\ (v > -1)\ S, \sigma[v=0]\rangle$$
$$\langle \texttt{while}\ (v > -1)\ S, \sigma[v=-2]\rangle \xrightarrow{A} \langle \epsilon, \sigma[v=-2]\rangle$$

2.9 Parallelism

In V^-, parallel composition is restricted to top level, so the statement is of the form

$$S_1 \parallel S_2 \parallel \cdots \parallel S_n$$

and each S_i does not contain any parallel operators. Some restrictions are necessary regarding composition of continuous assignments. First, a wire should not be assigned to in more than one processes. This is clear, because the meaning of a continuous assignment is that the wire will have the value of the assigned expression all the time, so if there are more than one continuous assignments to the same wire, unless the expressions have the same values (in this case, it is vacuous to have more than one assignment), there will be a contradiction. Secondly, there should be no circular continuous assignments to wires. For example, If we compose two continuous assignments `assign` $w_1 = w_2 + 1$ and `assign` $w_2 = w_1 + 1$, for example, the system will oscillate forever. One way to ensure that oscillation will not happen is to require that the constituent processes S_1, ..., S_n can be permuted as S_{i_1}, ..., S_{i_n} in such a way that if there is continuous assignment to a wire in process S_{i_j} then all the wires in the righthand side of the assignment cannot be assigned in any process S_{i_k} for $j < k \leq n$. Clearly, it is not possible to permute assignments `assign` $w_1 = w_2 + 1$ and `assign` $w_2 = w_1 + 1$ in this way: assume such a permutation exists, and let for example, `assign` $w_1 = w_2 + 1$ be before `assign` $w_2 = w_1 + 1$, but w_2 is in the righthand of the first assignment and is also assigned to in the second statement, thus a contradiction.

The execution of a parallel composition is modelled by interleaving transitions from constituent processes. A state transition of $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ is the result of one autonomous step of one process, possibly followed by the passive steps that triggered. A time step occurs when all the process are ready to advance the same amount of time.

$$\frac{\begin{array}{ll}\langle S_{i_1}, \sigma\rangle \xrightarrow{A} \langle S'_{i_1}, \sigma_1\rangle & \\ \left\langle S_{i_j}, \sigma_{j-1}\right\rangle \xrightarrow{P} \left\langle S'_{i_j}, \sigma_j\right\rangle & \text{for } 2 \leq j \leq k \\ \left\langle S_{i_j}, \sigma_k\right\rangle \stackrel{P}{\not\longrightarrow} \quad \text{for } k < j \leq n & \end{array}}{\langle S_1 \parallel S_2 \parallel \cdots \parallel S_n, \sigma\rangle \xrightarrow{A} \left\langle S'_{i_1} \parallel \cdots \parallel S'_{i_k} \parallel S_{i_{k+1}} \cdots \parallel S_{i_n}, \sigma_k\right\rangle}$$

where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$ that satisfy the above mentioned condition with respect to the order of updating of wires, and $\stackrel{P}{\not\longrightarrow}$ denotes that a passive step is not possible from the configuration. The process which initiates the transition is i_1, and it triggers $k-1$ passive transitions from processes $i_2, \ldots, i_k$. Note that in order to represent the transition rule, the order of processes may have been changed. It is easy to prove that $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ and $S_{i_1} \parallel S_{i_2} \parallel \cdots \parallel S_{i_n}$, where $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$, generates the same set of state sequences, and hence equivalent, so we can change the order of the processes arbitrarily, in particular in the original order, after applying the parallel composition rule.

$$\frac{\langle S_i, \sigma\rangle \stackrel{t}{\longrightarrow} \langle S_i', \sigma\rangle}{\langle S_1 \parallel S_2 \parallel \cdots \parallel S_n, \sigma\rangle \stackrel{t}{\longrightarrow} \langle S_1' \parallel \cdots \parallel S_n', \sigma\rangle}$$

We take the convention that $\epsilon \parallel S = S \parallel \epsilon$, and this can be easily generalised to the case of n processes.

The parallel composition rule is quite complex, so we shall study several examples. In these examples, the particular values of variables are not very important, so we shall denote the states by σ and σ_1 etc.

Example. Consider the program $v = v + 1 \;\parallel\; v = v + 2$. Both processes can take one autonomous step each

$$\langle v = v + 1, \sigma\rangle \stackrel{A}{\longrightarrow} \langle \epsilon, \sigma_1\rangle$$
$$\langle v = v + 2, \sigma\rangle \stackrel{A}{\longrightarrow} \langle \epsilon, \sigma_2\rangle$$

It follows from the transition rule that

$$\langle v = v + 1 \;\parallel\; v = v + 2, \sigma\rangle \stackrel{A}{\longrightarrow} \langle v = v + 2, \sigma_1\rangle$$
$$\langle v = v + 1 \;\parallel\; v = v + 2, \sigma\rangle \stackrel{A}{\longrightarrow} \langle v = v + 1, \sigma_2\rangle$$

Example. Consider the program $v = v + 1 \;\parallel\; \#2$. The second process can now do a time step. However, the only transition allowed by the composition is the discrete one, since a time step in the parallel composition needs the participation of all the processes

$$\langle v = v + 1 \;\parallel\; \#2, \sigma\rangle \stackrel{A}{\longrightarrow} \langle \#2, \sigma_1\rangle$$

Example. Consider the program `assign` $w_1 = -v \;\parallel\; \#2$. Now both processes can do a time step

$$\langle \texttt{assign}\ w_1 = -v, \sigma\rangle \stackrel{1}{\longrightarrow} \langle \texttt{assign}\ w_1 = -v, \sigma\rangle$$
$$\langle \#2, \sigma\rangle \stackrel{1}{\longrightarrow} \langle \#1, \sigma\rangle$$

It follows from the rule that

$$\langle \texttt{assign } w_1 = -v \ \parallel \ \#2, \sigma\rangle \xrightarrow{1} \langle \texttt{assign } w_1 = -v \ \parallel \ \#1, \sigma\rangle$$

Example. Consider the program

$$@(v)\ S \ \parallel \ v = v+1 \ \parallel \ v = v+2 \ \parallel \texttt{ assign } w_2 = w_1 + v \parallel \texttt{ assign } w_1 = -v$$

Since

$$\begin{array}{l}
\langle v = v+1, \sigma\rangle \xrightarrow{A} \langle \epsilon, \sigma_1\rangle \\
\langle @(v)\ S, \sigma_1\rangle \xrightarrow{P} \langle S, \sigma_1\rangle \\
\langle \texttt{assign } w_1 = -v, \sigma_1\rangle \xrightarrow{P} \langle \texttt{assign } w_1 = -v, \sigma_2\rangle \\
\langle \texttt{assign } w_2 = w_1 + v, \sigma_2\rangle \xrightarrow{P} \langle \texttt{assign } w_2 = w_1 + v, \sigma_3\rangle
\end{array}$$

applying the parallel rule, we have

$$\begin{array}{l}
\langle (@(v)\ S \ \parallel \ v = v+1 \ \parallel \ v = v+2 \ \parallel \texttt{ assign } w_2 = w_1 + v \parallel \texttt{ assign } w_1 = -v), \sigma\rangle \\
\xrightarrow{A} \langle \epsilon \ \parallel \ S \ \parallel \texttt{ assign } w_1 = -v \ \parallel \texttt{ assign } w_2 = w_1 + v \ \parallel \ v = v+2), \sigma_3\rangle
\end{array}$$

By our convention, we can rewrite the program after the transition as

$$S \ \parallel \ v = v+2 \ \parallel \texttt{ assign } w_2 = w_1 + v \parallel \texttt{ assign } w_1 = -v$$

3 Discussion

In this paper, we have given an operational semantics to V^-, a simple version of Verilog. The language contains interesting features like concurrency, timing behaviours, discrete actions as well as recursion. The semantics is given in the style of Plotkin's structural operational semantics. Operational semantics for concurrent languages can be found in, e.g., [1]. The use of labels to distinguish various transitions can be traced back to Aczel, who suggested this for defining semantics compositionally (cited e.g., in [2]). We have adopted the techniques, in particular to compose autonomous and passive transitions.

There has been some related work on formalising semantics of subsets of Verilog. Pace and He [6] have used an extension of Duration Calculus, an interval based real-time logic, to give a logical semantics to a subset of Verilog. We have also given a logical semantics to our subset using a different extension of Duration Calculus, which we believe models concurrency more faithfully [8]. Another approach is that of Gordon [4], which translates a V program into a normal form expressed in a simpler language.

Acknowledgements We thank He Jifeng for introducing Verilog to us and for suggesting studying its formalisation.

References

[1] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs (2nd edition)*. Springer–Verlag, 1997.

[2] W.-P. de Roever. The quest for compositionality. In *Proc. of IFIP Working Conf., The Role of Abstract Models in Computer Science*. Elsevier Science B.V. (North-Holland), 1985.

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

[4] M.J.C. Gordon. Synthesizable Verilog: sintax and semantics. June 1997.

[5] IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE std 1364-1995*, 1995.

[6] G.J. Pace and J.-F. He. Formal reasoning with Verilog HDL. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, 1998.

[7] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[8] G. Schneider and Q.-W. Xu. Towards a formal semantics of Verilog using duration calculus. In A. Ravn and H. Rischel, editors, *Formal Techniques for Real-Time and Fault Tolerant Systems (FTRTFT'98)*. LNCS, Springer-Verlag, 1998.