The Semantics of Verilog Using Transition System Combinators

Gordon J. Pace
gpace@cs.chalmers.se

Chalmers University of Technology, Sweden

Abstract. Since the advent of model checking it is becoming more common for languages to be given a semantics in terms of transition systems. Such semantics allow to model check properties of programs but are usually difficult to formally reason about, and thus do not provide a sufficiently abstract description of the semantics of a language. We present a set of transition system combinators that allow abstract and compositional means of expressing language semantics. These combinators are then used to express the semantics of a subset of the Verilog hardware description language. This approach allows reasoning about the language using both model checking and standard theorem proving techniques.

1 Introduction

Various benefits can be gained through the formal definition of the semantics of a language: documentation of the language semantics, enabling of formal reasoning, and automatic machine reasoning using techniques which allow (relatively) efficient automatic property checking. Giving a semantics through which we benefit from all these is usually difficult to achieve. For example, documenting the full semantics of an industrial strength language tends to yield inelegant semantics that are difficult to reason about, while aiming at automatic machine reasoning one tends to construct semantics which are impractical for interactive formal reasoning which may be necessary to verify large systems.

Hardware description languages, such as Verilog and VHDL have a very complex semantics defined in terms of their simulation behaviour. A lot of work has been done on the formalisation of these languages but most work manages to satisfactorily address only one of the desirable aspects of a formal semantics as listed earlier. Commercial tools for formal verification work only on register transfer level (RTL) descriptions. Furthermore, the semantics they use is usually not formally documented. In this paper, we identify a formal domain which allows us to document the semantics of these languages reasonably elegantly without sacrificing formal reasoning or model checking.

We are particularly interested in automatic machine verification through the use of model checking using BDDs [4], or SAT based techniques [1, 12]. The formal domain we use is a variant on standard transition systems since a wide variety

of techniques have been developed to check properties of transition systems efficiently and automatically. A number of transition system combinators permit us to express the semantics of languages compositionally and also allow effective reasoning about programs it the language.

To illustrate the effectiveness of this approach, we present the semantics of a subset of Verilog [11], which we call VeriSmall. The techniques used for VeriSmall readily scale up for larger subsets of Verilog. A more complete subset of behavioural Verilog has been formalised and implemented in a translator into SMV [8], details of which can be obtained from the author.

2 Notation

A relation between sets A and B is a set of pairs (a, b) where $a \in A$ and $b \in B$. The type of such a relation will be written as $A \leftrightarrow B$. If $\overset{A,B}{\mapsto}$ represents a relation of type $A \leftrightarrow B$, then $\overset{A,B}{\mapsto}_1 \cup \overset{A,B}{\mapsto}_2$ is the union of the two relations. The forward composition of the two relations will be written as $\overset{A,B}{\mapsto}; \overset{B,C}{\mapsto}$. $\overset{A,B}{\mapsto}^{-1}$ is the inverse of the relation — with type $B \leftrightarrow A$. The relational image of C ($C \subseteq A$) under $\overset{A,B}{\mapsto}$ is written as $\overset{A,B}{\mapsto}(C)$. The restriction of the domain of $\overset{A,B}{\mapsto}$ to set C is written as $\overset{A,B}{\mapsto} \restriction C$ and similarly, $\overset{A,B}{\mapsto} \not\restriction C$ is the restriction of the relation to the complement of C.

The overriding of $\overset{A,B}{\mapsto}_1$ by $\overset{A,B}{\mapsto}_2$, written as $\overset{A,B}{\mapsto}_1 \oplus \overset{A,B}{\mapsto}_2$ relates a with b if, either $a \overset{A,B}{\mapsto}_2 b$ or a is not in the domain of $\overset{A,B}{\mapsto}_2$ and $a \overset{A,B}{\mapsto}_1 b$. We make a relation total by adding any necessary identity transitions: $(\overset{A,A}{\mapsto})^{id} \overset{\text{def}}{=} id \oplus \overset{A,A}{\mapsto}$. Finally, $(\overset{A,B}{\mapsto}, \overset{C,D}{\mapsto})$ is the pairwise joining of inputs and outputs of the relations to get a relation of type $(A \times C) \leftrightarrow (B \times D)$.

3 VeriSmall

Verilog is a large and complex language. Documenting the detailed semantics of the whole language is beyond the scope of this paper. We thus work with a syntactic subset of Verilog, which we call VeriSmall. The sub-language is chosen such that the complexities of Verilog semantics are exposed. It is, however, sufficiently small to be presented in its entirety and reasoned about in this paper. The syntax of VeriSmall is given in figure 1.

VeriSmall is a concurrent language with steps made along parallel threads non-deterministically. The `#0` construct (read *zero delay*) blocks a thread until all other threads are finished or are similarly blocked. The behaviour of the rest of the language should be intuitively clear.

To illustrate the effect of zero delays, consider the program `(initial` $v = 1$ $\parallel$ `initial` $v = 0$`)`. Note that this program is non-deterministic and v can finish with either value 1 or 0. On the other hand, `(initial` $v = 1$ $\parallel$ `initial begin`

$$
\begin{array}{lll}
\langle \mathit{program} \rangle & ::= & \langle \mathit{module} \rangle \\
& | & \langle \mathit{program} \rangle \parallel \langle \mathit{program} \rangle \\
\langle \mathit{module} \rangle & ::= & \texttt{initial}\ \langle \mathit{statement} \rangle \\
& | & \texttt{always}\ \langle \mathit{statement} \rangle \\
\langle \mathit{statement} \rangle & ::= & \texttt{skip} \\
& | & \langle \mathit{variable} \rangle = \langle \mathit{expression} \rangle \\
& | & \texttt{wait (}\ \langle \mathit{variable} \rangle\ \texttt{)} \\
& | & \texttt{\#0}\ \langle \mathit{statement} \rangle \\
& | & \texttt{while (}\ \langle \mathit{expression} \rangle\ \texttt{)}\ \langle \mathit{statement} \rangle \\
& | & \texttt{if (}\ \langle \mathit{expression} \rangle\ \texttt{)}\ \langle \mathit{statement} \rangle\ \texttt{else}\ \langle \mathit{statement} \rangle \\
& | & \texttt{begin}\ \langle \mathit{block} \rangle\ \texttt{end} \\
\langle \mathit{block} \rangle & ::= & \langle \mathit{statement} \rangle \\
& | & \langle \mathit{block} \rangle\ ;\ \langle \mathit{block} \rangle
\end{array}
$$

Fig. 1. The syntax of VeriSmall

`#0` $v = 0$ `end)` is a deterministic program and always terminates with v carrying value 0.

Despite the fact that we have stripped Verilog down to its bare essentials, VeriSmall is still a substantially complex language.

3.1 The Simulation Cycle

The official documentation of Verilog describes the behaviour of a program in terms of how it should be interpreted by a simulator. This is also how we will informally describe the behaviour of VeriSmall programs.

VeriSmall is a concurrent language with a scheduling algorithm which manages the order of execution of concurrent threads. Each thread can be in one of three modes: enabled, delayed(0), waitfor(v) or finished. If a state is in mode waitfor(v) and the value of v is high, the thread is put into enabled mode. If no such threads exist, then the scheduler executes an enabled thread and the process is repeated. When no enabled threads are available, all threads in delayed(0) mode are made enabled and the process repeated from the beginning. The values of variables are also stored by the simulator. In VeriSmall, variables range over the values 1 (high), 0 (low), z (high impedance) and $\times$ (unknown). The simulation cycle is shown in figure 2.

The actions performed when moving along a thread depend on the instruction currently pointed at by the thread pointer.

Skip: The thread pointer is moved forward.

Assignments: If the first instruction is an assignment $v = e$, then the expression e is evaluated in the current state and v set to the value calculated.

Zero delays: To move along a thread pointing at `#0` P we set the thread's state to delayed(0) and move the thread pointer to P.

Conditionals: To move along the statement `if (`e`)` P `else` Q, e is evaluated and depending on its value, the thread pointer is moved to point at P or Q.

```
initialise all variables to ×
set all thread states to enabled
forever do
   if (there are threads waiting on a true variable) then
     set them enabled
   elsif (there are enabled threads) then
     choose one non-deterministically
     move one step along the thread
   elsif (any delayed by 0 modules) then
     enable all such modules
end
```

Fig. 2. VeriSmall simulation cycle

Wait: If the statement is `wait(`v`)` then the thread's state is set to waitfor(v) and the thread pointer is moved forward.

Loops: To move along the statement `while (`e`)` P, e is evaluated and depending on its value, the thread pointer is set to point at P or the first instruction after the loop instruction.

Blocks: The instruction pointer is moved to the first instruction in the block.

The top level module instructions are simply syntactic sugar, with `always` P corresponding to `while (1)` P and `initial` P to P.

4 Layered Transition Systems

Hardware description languages, the languages we are mainly interested in expressing the semantics of, usually have an inherent concept of priority of execution. In the case of VeriSmall, for instance, threads blocked by a zero delay get lower priority than enabled ones.

With this specification in mind, we add extra priority information by placing all states in one of an ordered set of layers — the higher the the layer, the higher priority given to that state when composing systems in parallel. We also use standard transition system combinators such as union and sequential composition, as intermediate language constructs.

It is important to note that the layering information is only necessary to compose systems and can be hidden away when we want to check properties of a transition system.

4.1 Formal Definition

Since we would like to be able to compare the layering information in one machine to that in another, we will assume the existence of a fixed set of layers $LAYER$ over which the total ordering $\geq$ is defined.

Definition 1: A *finite layered transition system* (henceforth FLTS), is a 5-tuple $\langle Q, I, F, \mapsto, \mathsf{layer}\rangle$, where:

Q	finite set of states,
I	set of initial states ($I \subseteq Q$),
F	set of final states ($F \subseteq Q$),
$\mapsto$	transition relation between states
layer	total function from states to layers

We will use $\mathcal{M}$ to represent arbitrary FLTS. If necessary, we will also use subscripts. Unless otherwise stated, $\mathcal{M}$ is the FLTS $\langle Q, I, F, \mapsto, \mathsf{layer}\rangle$ and $\mathcal{M}_i$ is $\langle Q_i, I_i, F_i, \overset{i}{\mapsto}, \mathsf{layer}_i\rangle$.

4.2 Transition System Combinators

Definition 2: Given a FLTS $\mathcal{M}$ and a predicate $r_i : Q \to \mathsf{bool}$, then $\mathcal{M}$ *domain restricted to* r_i, is defined by:

$$r_i \lhd \mathcal{M} \overset{\text{def}}{=} \langle Q, \{\sigma : I \mid r_i(\sigma)\}, F, \mapsto, \mathsf{layer}\rangle$$

Definition 3: Given a FLTS $\mathcal{M}$ and a predicate $r_f : Q \to \mathsf{bool}$, then $\mathcal{M}$ *range restricted to* r_i, is defined by:

$$\mathcal{M} \rhd r_f \overset{\text{def}}{=} \langle Q, I, \{\sigma : F \mid r_f(\sigma)\}, \mapsto, \mathsf{layer}\rangle$$

Definition 4: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$, we define the *union* of the two systems $\mathcal{M}_1 \cup \mathcal{M}_2$ as follows:

$$\mathcal{M}_1 \cup \mathcal{M}_2 \overset{\text{def}}{=} \langle Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \overset{1}{\mapsto} \cup \overset{2}{\mapsto}, \mathsf{layer}_1 \cup \mathsf{layer}_2\rangle$$

Note that the two transition systems *must* agree on the layer of any common states.

Definition 5: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$ and a relation jn between the final states of $\mathcal{M}_2$ and the initial of $\mathcal{M}_1$ ($jn : F_1 \leftrightarrow I_2$), we define the *catenation* of the two systems $\mathcal{M}_1 \overset{jn}{;} \mathcal{M}_2$:

$$\mathcal{M}_1 \overset{jn}{;} \mathcal{M}_2 \overset{\text{def}}{=} \langle Q_1 \cup Q_2,\ \ jn^{id}(I_1), F_2,\ \ \overset{1}{\mapsto}; jn^{id} \cup \overset{2}{\mapsto},\ \ \mathsf{layer}_1 \cup \mathsf{layer}_2\rangle$$

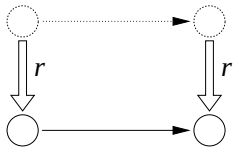
Thus, whenever a transition can take us from a state σ to state σ' which is related (via jn) to σ'', we add the direct transition from σ to σ''. States in $\mathcal{M}_2$ related to initial states in $\mathcal{M}_1$ are also initial.

Since we have catenation, we can similarly define the reflexive transitive closure of a FLTS.

Definition 6: Given a FLTS $\mathcal{M}$ and a relation jn between the final states and initial states ($jn : F \leftrightarrow I$, with disjoint domain and range) we define the *reflexive transitive closure* of $\mathcal{M}$ — written as $\mathcal{M}^*$ — as follows:

$$\mathcal{M}^* \overset{\text{def}}{=} \langle Q, jn^{id}(I), F \setminus \mathrm{dom}(jn), \mapsto; jn^{id}, \mathsf{layer}\rangle$$

Given a FLTS we sometimes need to re-map the states — this can be seen as a form of data abstraction. Given a relation r between the old and the new states, we can use this relation to create a new FLTS such that there is a transition between two new states σ and σ' if and only if they are related (by r) to two old states between which there was a transition.



Definition 7: Given a FLTS $\mathcal{M}$ and relation $r : Q \leftrightarrow Q'$ we can define the *remapping of* $\mathcal{M}$ *with* r, written as $r(\mathcal{M})$, as follows:

$$r(\mathcal{M}) \stackrel{\text{def}}{=} \langle\ Q', r(I), r(F), r^{-1}; \mapsto; r,\ \ \lambda\sigma \cdot \max(r^{-1}; \mathsf{layer}(\sigma))\ \rangle$$

Note that in some cases, r may relate several old states with a new one. In this case the new state assumes the highest priority of its related states.

4.3 Parallel Composition

To compose two systems in parallel we need to define what the result of joining two states is. It is tempting to always take the Cartesian product of the old state spaces to be the new state space. However, this will complicate matters when we have states with overlapping domains. We thus choose to explicitly express this operation as two relations $\succ_1$ and $\succ_2$ which join two states together giving priority to the first and second state respectively.

$$\succ_1, \succ_2 \colon (Q_1 \times Q_2) \to Q$$

When we do not care which state is given priority, we will use $\succ$ defined as $\succ_1 \cup \succ_2$.

We will also need to decompose states into separate parts:

$$\prec \colon Q \to (Q_1 \times Q_2)$$

Note that all the following FLTS combinators are parametrised by these functions.

Given two FLTS, we can construct a FLTS acting like the coupling of the two FLTS — this corresponds to running the two systems together, where for every transition one system makes, the other also performs exactly one transition.

Definition 8: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$, we can define the *synchronous parallel composition* of the two systems $\mathcal{M}_1 \| \mathcal{M}_2$ as follows:

$$\begin{aligned}\mathcal{M}_1 \| \mathcal{M}_2 \stackrel{\text{def}}{=} \langle\ & \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (\overset{1}{\mapsto}, \overset{2}{\mapsto}); \succ, \prec; (\mathsf{layer}_1, \mathsf{layer}_2); \max_2\ \rangle\end{aligned}$$

(where $\max_2$ relates a pair of numbers with the maximum of the two)

We will write this transition relation as $(\overset{1}{\mapsto} \parallel \overset{2}{\mapsto})$.

Sometimes it is necessary to allow the processes to stutter – while one of the processes carries out a transition, the other remains in the same state.

Definition 9: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$, we define the *parallel composition with stuttering* of the two systems $\mathcal{M}_1 \|^\iota \mathcal{M}_2$ as follows:

$$\begin{aligned}\mathcal{M}_1 \|^\iota \mathcal{M}_2 \stackrel{\text{def}}{=} \langle\ & \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (\overset{1}{\mapsto} \cup id, \overset{2}{\mapsto}); \succ_2 \cup \prec; (\overset{1}{\mapsto}, \overset{2}{\mapsto} \cup id); \succ_1, \\ & \prec; (\mathsf{layer}_1, \mathsf{layer}_2); \max_2\ \rangle\end{aligned}$$

We will write this composition of transition relations as $(\overset{1}{\mapsto} \|^{\iota} \overset{2}{\mapsto})$.

Sometimes we would like to compose two systems concurrently, with only one system performing a transition at a time.

Definition 10: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$, we define the *interleaving* of the two systems $\mathcal{M}_1 \lVert \mathcal{M}_2$ as follows:

$$\begin{aligned}\mathcal{M}_1 \lVert \mathcal{M}_2 \stackrel{\text{def}}{=} \langle\ & \succ (S_1 \times S_2), \succ (I_1 \times I_2), \succ (F_1 \times F_2), \\ & \prec; (id, \overset{2}{\mapsto}); \succ_2 \cup \prec; (\overset{1}{\mapsto}, id); \succ_1, \\ & \prec; (\mathsf{layer}_1, \mathsf{layer}_2); \max_2\ \rangle\end{aligned}$$

We will write this composed transition relation as $(\overset{1}{\mapsto}\ \lVert\ \overset{2}{\mapsto})$.

4.4 Layered Parallel Composition

Definition 11: Given two FLTS $\mathcal{M}_1$ and $\mathcal{M}_2$ and a function which determines how the layers will be composed: $+ : LAYER \to \{\|, \|^{\iota}, \lVert\}$, we define the *layered parallel composition* of the two systems $\mathcal{M}_1 + \mathcal{M}_2$ as follows:

$$\begin{aligned}\mathcal{M}_1 + \mathcal{M}_2 \stackrel{\text{def}}{=} \langle\ & \succ (S_1 \times S_2), \\ & \succ (I_1 \times I_2), \succ (F_1 \times F_2), \mapsto, \\ & \prec; (\mathsf{layer}_1, \mathsf{layer}_2); \max_2\ \rangle\end{aligned}$$

where

$$\begin{aligned}\sigma \mapsto \sigma' \stackrel{\text{def}}{=}\ & \mathsf{layer}(\prec \sigma)_1 > \mathsf{layer}(\prec \sigma)_2 \wedge \sigma(\overset{1}{\mapsto}\ \lVert\ \emptyset)\sigma'\ \vee \\ & \mathsf{layer}(\prec \sigma)_1 < \mathsf{layer}(\prec \sigma)_2 \wedge \sigma(\emptyset\ \lVert\ \overset{2}{\mapsto})\sigma'\ \vee \\ & \mathsf{layer}(\prec \sigma)_1 = \mathsf{layer}(\prec \sigma)_2 = i \wedge \sigma(\overset{1}{\mapsto} +_i \overset{2}{\mapsto})\sigma'\end{aligned}$$

This means that if one of the state components has a higher priority, the other component is not allowed to perform any transitions (lines 1 and 2) while, if both components have the same priority, then the transition relation is determined by the layer in which they lie (line 3).

4.5 FLTS Combinator Expressions

Definition 12: We define $TS_{\equiv}$ to be the language of all valid combinator expressions over layered transition systems with basic transition systems as terminals. $TS^{V}_{\equiv}$ is the similar language but which may also have variables (elements of V) as terminals.

A *context* $C(X)$ is an expression in $TS^{\{X\}}_{\equiv}$. Given an expression $\mathcal{M} \in TS_{\equiv}$, $C(\mathcal{M})$ is the same as $C(X)$, but with instances of X replaced by $\mathcal{M}$.

5 Laws of FLTS

The FLTS combinators presented in the previous section allow us to specify the semantics of VeriSmall and similar languages in a concise and readable manner. They also satisfy a number of laws which will enable us to reason about VeriSmall programs more easily. In this section we present a number of such laws.

5.1 Reachability and Inclusion

In this paper the only semantic property of FLTS we are concerned with is reachability:

Definition 13: Given a FLTS $\mathcal{M}$, we define the set of reachable states from states Σ as follows:

$$\mathsf{reachable}_{\mathcal{M}}(\Sigma) \stackrel{\text{def}}{=} \mapsto^* (\Sigma)$$

If we only care about the reachable states in a particular FLTS (starting from the initial states), we write this as:

$$\mathsf{reachable}(\mathcal{M}) \stackrel{\text{def}}{=} \mathsf{reachable}_{\mathcal{M}}(I)$$

Given a projection function $\pi : (Q_1 \cup Q_2) \to Q$, we say that σ and σ' are π-equivalent if $\pi(\sigma) = \pi(\sigma')$. We write this as $\sigma =_\pi \sigma'$. Similarly, we can extend this notation to sets of states:

$$\Sigma \subseteq_\pi \Sigma' \stackrel{\text{def}}{=} \pi(\Sigma) \subseteq \pi(\Sigma')$$

We will define FLTS containment with respect to a transition function.

Definition 14: A FLTS $\mathcal{M}_1$ is said to be *contained in* $\mathcal{M}_2$ *with respect to a projection* π, written as $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_2$, if:

- States are contained under π: $Q_1 \subseteq_\pi Q_2$
- Initial states are contained under π: $I_1 \subseteq_\pi I_2$
- Final states are inversely contained under π: $F_2 \subseteq_\pi F_1$
- π-equivalent elements in Q_1 and in Q_2 belong to the same layer:
  $$\forall \sigma : Q_1,\ \sigma' : Q_1 \cdot \sigma =_\pi \sigma' \implies \mathsf{layer}_1(\sigma) = \mathsf{layer}_2(\sigma')$$
  This is equivalent to: $\pi^{-1}; \mathsf{layer}_1 = \pi^{-1}; \mathsf{layer}_2$
- A member of Q_2 must be able to emulate (up to π) all transitions of π-equivalent members of Q_1:
  $$\forall (\sigma_1, \sigma_1') :\stackrel{1}{\mapsto}, \sigma_2 : Q_2 \cdot \sigma_1 =_\pi \sigma_2 \implies \exists \sigma_2' : Q_2 \cdot \sigma_2 \stackrel{2}{\mapsto} \sigma_2' \wedge \sigma_1' =_\pi \sigma_2'$$
  This is equivalent to: $\pi; \pi^{-1}; \stackrel{1}{\mapsto} \subseteq \stackrel{2}{\mapsto}; \pi; \pi^{-1}$

Transition system inclusions guarantee containment of reachable states.

Lemma 1: If $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_2$ then, for any set of states Σ, $\mathsf{reachable}_{\mathcal{M}_1}(\Sigma) \subseteq_\pi \mathsf{reachable}_{\mathcal{M}_2}(\Sigma)$.

5.2 Monotonicity

Provided that π obeys a number of properties (dictating how it distributes over the relations used in FLTS construction) expressions in $TS_\equiv$ are monotone with respect to $\sqsubseteq_\pi$. These proofs follow almost exclusively from the monotonicity of relational mapping, relational composition and set union. A frequently used property of functions is $\pi; \pi^{-1}; \pi = \pi$.

Lemma 2: If $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_3$ and $\mathcal{M}_2 \sqsubseteq_\pi \mathcal{M}_4$ then:

i. Reduction of initial states: If $i = \pi; i$ then $i \triangleleft \mathcal{M}_1 \sqsubseteq_\pi i \triangleleft \mathcal{M}_3$.

ii. Reduction of final states: If $f = \pi; f$ then $\mathcal{M}_1 \triangleright f \sqsubseteq_\pi \mathcal{M}_3 \triangleright f$.

iii. Remapping: If $r; \pi; \pi^{-1} = \pi; \pi^{-1}; r$ then $r(\mathcal{M}_1) \sqsubseteq_\pi r(\mathcal{M}_3)$.

iv. Union: $\mathcal{M}_1 \cup \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \cup \mathcal{M}_4$.

v. Sequential composition: If $jn^{id}; \pi; \pi^{-1} = \pi; \pi^{-1}; jn^{id}$ then $\mathcal{M}_1 \overset{jn}{;} \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \overset{jn}{;} \mathcal{M}_4$.

vi. Reflexive transitive closure: If $jn^{id}; \pi; \pi^{-1} = \pi; \pi^{-1}; jn^{id}$ and if also $\sigma =_\pi \sigma' \implies (\sigma \in \text{dom } jn \Leftrightarrow \sigma' \in \text{dom } jn)$, then $\mathcal{M}_1^* \sqsubseteq_\pi \mathcal{M}_2^*$.

vii. Synchronous parallel composition: If the relation $\pi; \pi^{-1}$ commutes with state composition and state decomposition:

$$\begin{aligned} \succ; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ \\ \prec; (\pi; \pi^{-1}, \pi; \pi^{-1}) &= \pi; \pi^{-1}; \prec \end{aligned}$$

then $\mathcal{M}_1 \| \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \| \mathcal{M}_4$.

viii. Parallel composition with stuttering: Similarly, if $\pi; \pi^{-1}$ commutes with state composition and state decomposition:

$$\begin{aligned} \succ_1; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ_1 \\ \succ_2; \pi; \pi^{-1} &= (\pi; \pi^{-1}, \pi; \pi^{-1}); \succ_2 \\ \pi; \pi^{-1}; \prec &= \prec; (\pi; \pi^{-1}, \pi; \pi^{-1}) \end{aligned}$$

then $\mathcal{M}_1 \|' \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \|' \mathcal{M}_4$.

ix. Interleaving: Under the same constraints as case viii, $\mathcal{M}_1 \between \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 \between \mathcal{M}_4$.

x. Layered parallel composition: Under the same constraints as case viii, $\mathcal{M}_1 + \mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_2 + \mathcal{M}_4$.

Proof: The proofs of the different cases are very similar and thus, we just present the proof of case ix.

States:

π(states of $\mathcal{M}_1 \between \mathcal{M}_3$)
= { definition of interleaving }
$\succ; \pi(Q_1 \times Q_3)$
= { $\pi = \pi; \pi^{-1}; \pi$ }
$\succ; \pi; \pi^{-1}; \pi(Q_1 \times Q_3)$
= { assumption about π and $\succ$ }
$(\pi; \pi^{-1}, \pi; \pi^{-1}); \succ; \pi(Q_1 \times Q_3)$
$\subseteq$ { $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_3$ and $\mathcal{M}_2 \sqsubseteq_\pi \mathcal{M}_4$ }
$(\pi; \pi^{-1}, \pi; \pi^{-1}); \succ; (\pi(Q_2) \times \pi(Q_4))$
= { refold back }
$\succ; \pi(Q_2 \times Q_4)$
= { definition of interleaving }
π(states of $\mathcal{M}_2 \between \mathcal{M}_4$)

Transitions:

$\pi; \pi^{-1}$; (transition relation of $\mathcal{M}_1 \between \mathcal{M}_3$)
= { definition of interleaving }
$\pi; \pi^{-1}; \prec; (id, \overset{3}{\mapsto}); \succ_2 \cup$
$\pi; \pi^{-1}; \prec; (\overset{1}{\mapsto}, id); \succ_1$
= { assumptions about π }
$\prec; (\pi; \pi^{-1}; id, \pi; \pi^{-1}; \overset{3}{\mapsto}); \succ_2 \cup$
$\prec; (\pi; \pi^{-1}; \overset{1}{\mapsto}, \pi; \pi^{-1}; id); \succ_1$
$\subseteq$ { $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_2$ and $\mathcal{M}_3 \sqsubseteq_\pi \mathcal{M}_4$ }
$\prec; (\pi; \pi^{-1}; id, \overset{4}{\mapsto}; \pi; \pi^{-1}); \succ_2 \cup$
$\prec; (\overset{2}{\mapsto}; \pi; \pi^{-1}, \pi; \pi^{-1}; id); \succ_1$
= { refold back }
$\prec; (id, \overset{4}{\mapsto}); \succ_2; \pi; \pi^{-1} \cup$
$\prec; (\overset{2}{\mapsto}, id); \succ_1; \pi; \pi^{-1}$
= { definition of interleaving }
(transition relation of $\mathcal{M}_2 \between \mathcal{M}_4$); $\pi; \pi^{-1}$

Initial and final states information and the layer information proofs proceed similar to the ones above.

□

Theorem 1: Combinator contexts are monotonic. If $\mathcal{M}_1 \sqsubseteq_\pi \mathcal{M}_2$ then $C(\mathcal{M}_1) \sqsubseteq_\pi C(\mathcal{M}_2)$.

Proof: This follows directly using structural induction and lemma 2. □

6 Formal Semantics of VeriSmall

We now document the formal semantics of VeriSmall in terms of FLTS. Note that the use of layers allows us to avoid having worry about the details of the complex simulation cycle when describing the semantics of sequential programs.

The state of the transition system corresponds to the state of the simulator: the values stored by variables, the position in the different threads and the state of each thread.

The state will be be a store: a 'function' from variable names to values[1]. $\sigma(v)$ is the value of variable in state σ. We extend this notation to expressions. Thus, for example, $\sigma(e \texttt{ and } f)$ will be defined to be $\sigma(e) \texttt{ and } \sigma(f)$. Special variable names *pos* and *grd* (are used to represent the position in the thread and its state respectively. σ_{var} is the store restricted to Verilog variables (that is excluding the position and state variables).

$\sigma[v := e]$ is the state just like σ but with the value of variable v changed to $\sigma(e)$. The set of all states with Verilog variables in V, the value of *pos* ranging over P and that of *grd* over G will be referred to as $\Sigma_{V,P,G}$.

In the course of the language semantics definition, we will sometimes need to know the size of a statement or statement block. Given a statement P we can define size(P) using primitive recursion over the structure of P:

$$
\begin{array}{llll}
\text{size}(\texttt{skip}) & \stackrel{\text{def}}{=} 1 & \text{size}(P;Q) & \stackrel{\text{def}}{=} \text{size}(P) + \text{size}(Q) \\
\text{size}(v = e) & \stackrel{\text{def}}{=} 1 & \text{size}(\texttt{begin } P \texttt{ end}) & \stackrel{\text{def}}{=} 1 + \text{size}(P) \\
\text{size}(\texttt{wait (}v\texttt{)}) & \stackrel{\text{def}}{=} 1 & \text{size}(\texttt{\#0 } P) & \stackrel{\text{def}}{=} 1 + \text{size}(P) \\
\text{size}(\texttt{if } e \texttt{ } P \texttt{ else } Q) & \stackrel{\text{def}}{=} 1 + \text{size}(P) & \text{size}(\texttt{while } e \texttt{ } P) & \stackrel{\text{def}}{=} 1 + \text{size}(P) \\
 & \quad + \text{size}(Q) & &
\end{array}
$$

It can be proved that the value of *pos* in any state of the transition system produced from a sequential program P will never reach or exceed size(P).

Similarly we can define the function vars(x) which returns the set of variables occuring in program or expression x.

Layers and Other Preliminaries The subset of Verilog we present uses four layers: unblocking states waiting on a variable takes highest priority, followed by transitions on enabled threads are of highest priority, then threads delayed by zero time and finally, the finished threads. Thus:

[1] The quotes around the word function are there because different variables in the domain of a store may return values of different types and therefore it is not, strictly speaking, a function.

$$LAYER \stackrel{\text{def}}{=} \{\text{FIRE}, \text{ENA}, \text{DEL}_0, \text{FIN}\}$$
$$\text{FIRE} > \text{ENA} > \text{DEL}_0 > \text{FIN}$$

These layer names will also be overloaded with the related constant function — for example ENA will also be used for $\lambda\sigma \cdot \text{ENA}$.

We will often need to combine together different automata but making sure that the states are disjoint. This will be done by modifying the *pos* values using a remapping of the FLTS. If pos_{+n} is the function $pos_{+n}(\sigma) = \sigma[pos := pos + n]$, then the remapping $pos_{+n}(\mathcal{M})$ is the renaming we require.

Each FLTS produced will handle a particular set of variables. To increase the set of variables handled by a transition system we use interleaving composition. If S is $\Sigma_{V,\emptyset,\emptyset}$ then

$$add_V(\mathcal{M}) \stackrel{\text{def}}{=} \langle S, S, S, \emptyset, \text{FIN}\rangle \parallel\!\parallel \mathcal{M}$$

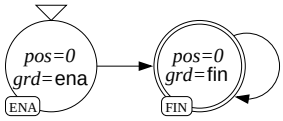
The relations used for this composition are the minimal relations satisfying:

$$\begin{array}{l}\sigma \qquad\quad \prec\ (\sigma \restriction V, \sigma \not\restriction V)\\ (\sigma_1, \sigma_2) \succ_1 \sigma_2 \oplus \sigma_1\\ (\sigma_1, \sigma_2) \succ_2 \sigma_1 \oplus \sigma_2\end{array}$$

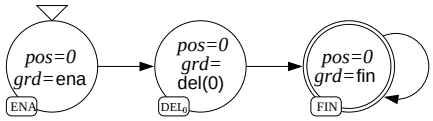
Expressions such as $add_V(pos_{+n}(\mathcal{M}))$ will be written as $\mathcal{M}_{V,+n}$.

The Semantics

Skip: The interpretation of `skip`, $[\![\texttt{skip}]\!]$ is a FLTS with two states:



Zero delays: Note that all zero delays are followed by a statement. However, this simply corresponds to normal sequential composition and we can give the semantics of zero delays as independent statements and then define $[\![\texttt{\#0}\ P]\!]$ to be $[\![\texttt{\#0;}P]\!]$. The semantics of zero delays $[\![\texttt{\#0}]\!]$ is rather similar to that of `skip`:



Assignments: $[\![v = e]\!]$ is a FLTS with the following set of states:

$$Q = \Sigma_{\text{vars}(v=e),\{0\},\{\mathsf{enabled},\mathsf{finished}\}}$$

The initial states are those in which the thread is enabled:

$$\{\sigma : Q \mid \sigma(grd) = \mathsf{enabled}\}$$

The final states are those in which the thread is finished:

$$\{\sigma : Q \mid \sigma(grd) = \mathsf{finished}\}$$

The system can go from enabled states to finished ones by setting the value of the variable to that of the expression.

$$\sigma[grd := \mathsf{enabled}] \mapsto \sigma[v := e][grd := \mathsf{finished}]$$

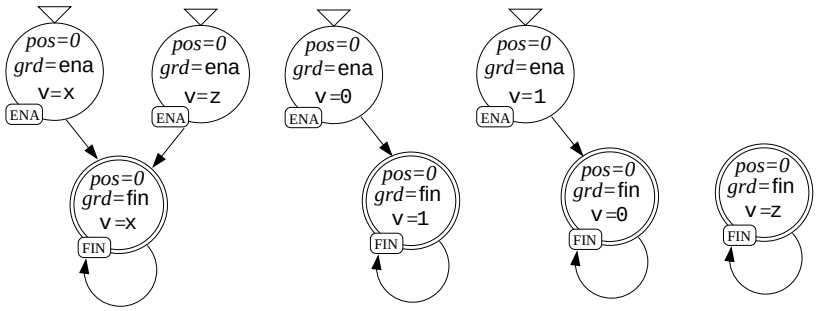
Finished states can perform reflexive transitions:

$$\sigma[grd := \mathsf{finished}] \mapsto \sigma[grd := \mathsf{finished}]$$

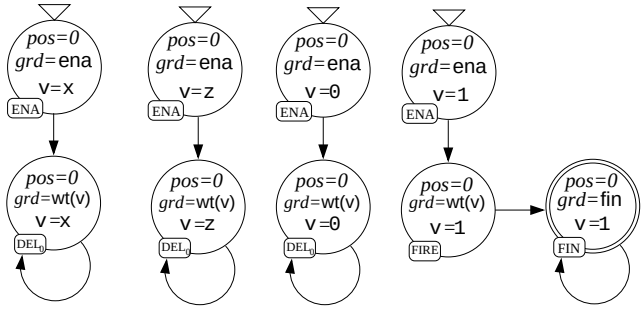
The layer of a state can be deduced from its guard:

$$\mathsf{layer}(\sigma) = \begin{cases} \text{ENA if } \sigma(grd) = \mathsf{enabled} \\ \text{FIN otherwise} \end{cases}$$

Below is the FLTS $[\![\texttt{v =!v}]\!]$:



Wait: The semantics of `wait(`v`)` are given by the FLTS in the diagram below:



The first step the simulator may perform on such a thread is setting the guard to $\mathsf{waitfor}(v)$ and keep the variable value constant. States guarded by $\mathsf{waitfor}(v)$ and in which v is not high can only remain the same state, but if v is high, the system can proceed to terminate.

Assumptions: To make the presentation of the semantics clearer, we introduce a new statement in VeriSmall:

$$\langle \mathit{statement} \rangle ::= \langle \mathit{expression} \rangle^{\top}$$

$e^{\top}$, read *assume* e, moves along the thread if e evaluates to 1 but aborts the thread if not. To complete the definition of the size function: $\mathrm{size}(e^{\top}) \stackrel{\text{def}}{=} 0$. The set of states is:

$$Q = \{\sigma : \Sigma_{\mathrm{vars}(e),\{0\},\{\mathsf{finished}\}} \mid \sigma(e) = 1\}$$

The semantics of assumption:

$$[\![e^{\top}]\!] \stackrel{\text{def}}{=} \langle Q, Q, Q, id, \text{FIN} \rangle$$

Sequential composition: $[\![P; Q]\!]$ can be expressed in terms of the FLTS $[\![P]\!]$ and $[\![Q]\!]$.

$$[\![P; Q]\!] \stackrel{\text{def}}{=} [\![P]\!]_{\mathrm{vars}(Q)} \stackrel{jn}{;} [\![Q]\!]_{\mathrm{vars}(P),+\mathrm{size}(P)}$$

jn relates states with the same store: $\sigma(jn)\sigma'$ if and only if $\sigma_{var} = \sigma'_{var}$ (and $\sigma \in F_1$ and $\sigma' \in I_2$).

Blocks: It is tempting to define the semantics of a block simply by removing the outer `begin end` keywords. Note, however, that the simulator takes one cycle to proceed to the first instruction. An accurate description of the semantics is thus:

$$[\![\texttt{begin } P \texttt{ end}]\!] \stackrel{\text{def}}{=} [\![\texttt{skip; } P]\!]$$

Conditionals: The semantics of conditionals can be expressed using restriction of initial states. However, the semantics are slightly more complex due to the fact that the simulator takes one cycle to evaluate the condition:

$$\begin{array}{l}[\![\texttt{if (}e\texttt{) } P \texttt{ else } Q]\!] \stackrel{\text{def}}{=} \\ [\![\texttt{skip}]\!]_V \overset{jn}{;} \\ \left(\begin{array}{l} [\![(e=1)^\top; P]\!]_{\text{vars}(Q),+1} \cup \\ [\![(e\neq 1)^\top; Q]\!]_{\text{vars}(P),+1+\text{size}(P)} \end{array} \right) \end{array}$$

where $V = \text{vars}(\texttt{if (}e\texttt{) } P \texttt{ else } Q)$ and the jn relation is the same as the one used in sequential composition.

While loops: The semantics of while loops are similar:

$$\begin{array}{l}[\![\texttt{while (}e\texttt{) } P]\!] \stackrel{\text{def}}{=} \\ [\![\texttt{skip}]\!]_V \overset{jn}{;} \\ ([\![(e=1)^\top; P; \texttt{skip}]\!]^* \cup [\![(e\neq 1)^\top]\!]_{\text{vars}(P)}) \end{array}$$

Again note that the simulator takes a cycle to evaluate the expression and that the jn relation is the same as the one used in sequential composition.

Parallel composition: Parallel composition can now be defined in terms of layered composition:

$$[\![P \| Q]\!] \stackrel{\text{def}}{=} [\![P]\!] + [\![Q]\!]$$

This layered composition uses:

$$+_{\text{FIRE}} = \| \quad +_{\text{ENA}} = \|\!\| \quad +_{\text{DEL}_0} = \| \quad +_{\text{FIN}} = \|$$

The state constructor relations are:

$$\begin{array}{rl}(\sigma, \sigma') \succ_1 \sigma'_{vars} \oplus \sigma_{vars} \oplus \{ & pos \mapsto (\sigma(pos), \sigma'(pos)), \\ & grd \mapsto (\sigma(grd), \sigma'(grd))\} \\ (\sigma, \sigma') \succ_2 \sigma_{vars} \oplus \sigma'_{vars} \oplus \{ & pos \mapsto (\sigma(pos), \sigma'(pos)), \\ & grd \mapsto (\sigma(grd), \sigma'(grd))\} \end{array}$$

The state decomposition relation is simply the inverse of these two: $\prec \stackrel{\text{def}}{=} \succ^{-1}$.

7 Reasoning about VeriSmall

Finally, we show how these semantics can be used to prove general theorems about programs. These may later be used to aid or simplify automatic verification of programs.

Programs are usually built in different parts which are eventually joined together. It is therefore important to be able to show that individual parts satisfy certain properties whatever is plugged into them. The question thus arises: How can we prove such properties of our programs using model checking?

The scenario depicted in the previous paragraph corresponds to a program being a context with variables pointing out where other programs are to be plugged

in. Thus, for example, one may be given the program context $C(P)$: `initial begin x=0;` P `; x=!y end` and be asked to prove that if P is a program using only variable `y`, then `x` and `y` are never high at the same time.

The technique we use is to substitute P with the most non-deterministic program possible and prove the property of this new program. If we can prove that:

$$\sigma \in \mathsf{reachable}([\![C(\texttt{chaos}(\{y\}))]\!]) \implies \sigma(\texttt{x} \neq 1 \vee \texttt{y} \neq 1)$$

then it should follow that for any program P which has alphabet $\{\texttt{y}\}$:

$$\sigma \in \mathsf{reachable}([\![C(P)]\!]) \implies \sigma(\texttt{x} \neq 1 \vee \texttt{y} \neq 1)$$

Note that if we define the FLTS semantics of `chaos(`V`)`, the first statement can be checked automatically using standard reachability techniques.

7.1 Chaos

The semantics of `chaos(`V`)`, where V is a set of variables, is straightforward to define.

The states are: $Q = \Sigma_{V,\{0\},\{\mathsf{enabled},\mathsf{delayed}(0),\mathsf{finished}\}}$

The initial states are those enabled: $\{\sigma : Q \mid \sigma(grd) = \mathsf{enabled}\}$

The final states are those which are finished: $\{\sigma : Q \mid \sigma(grd) = \mathsf{finished}\}$

States can do anything, but once finished they must remain so:

$$id \cup ((Q \setminus F) \times Q)$$

The layer can be deduced from the mode:

$$\mathsf{layer}(\sigma) = \begin{cases} \text{ENA} & \text{if } \sigma(grd) = \mathsf{enabled} \\ \text{DEL}_0 & \text{if } \sigma(grd) = \mathsf{delayed}(0) \\ \text{FIN} & \text{if } \sigma(grd) = \mathsf{finished} \end{cases}$$

Also, $\text{size}(\texttt{chaos(}V\texttt{)}) \stackrel{\text{def}}{=} 1$ and $\text{vars}(\texttt{chaos(}V\texttt{)}) \stackrel{\text{def}}{=} V$

7.2 The Theorem

It is quite easy to formulate the result we desire incorrectly. For example, in the example given earlier, the safety condition that the thread position never exceeds 4 can be proved of $C(\texttt{chaos(}V\texttt{)})$ but this will not be true for long enough instances of P. It is thus important to restrict safety conditions to variables and guards. Similarly we must make sure that P uses no variables other than those in V.

$$\frac{\begin{array}{l}\text{vars}(P) = V \\ \forall \sigma \cdot \sigma(prop) = (red; \sigma)(prop) \\ \forall \sigma \cdot \sigma \in \mathsf{reachable}([\![C(\texttt{chaos(}V\texttt{)})]\!]) \\ \qquad \implies \sigma(prop)\end{array}}{\forall \sigma \cdot \sigma \in \mathsf{reachable}([\![C(P)]\!]) \implies \sigma(prop)}$$

where $red(\sigma) \stackrel{\text{def}}{=} (\sigma[grd := \text{if } (grd = \mathsf{waitfor}(v)) \text{ then } grd' \text{ else } grd]) \not\downarrow pos$, and $grd' = (\mathsf{layer}(\sigma) = \text{FIRE})$ then $\mathsf{enabled}$ else $\mathsf{delayed}(0)$.

Lemma 3: If $\text{vars}(P) = V$, then $[\![P]\!] \sqsubseteq_{red} [\![\texttt{chaos(}V\texttt{)}]\!]$.

Lemma 4: VeriSmall programs are monotone with inclusions with variables projections: If $[\![P]\!] \sqsubseteq_{red} [\![Q]\!]$ then $[\![C(P)]\!] \sqsubseteq_{red} [\![C(Q)]\!]$.

This result follows by checking that all relations used in the semantics commute with *red* as specified in lemma 2, and using structural induction over the program context C.

From lemmata 3 and 4, it follows that $[\![C(P)]\!] \sqsubseteq_{red} [\![C(\texttt{chaos(}V\texttt{)})]\!]$. The desired result then follows from lemma 1.

8 A Small Example

The following example shows how the techniques shown in this paper can be applied to a small example. Consider the following VeriSmall program:

```
initial begin v=0; P1; v=1; wait(w); P2; end;
initial begin w=0; Q1; w=1; wait(v); Q2; end;
```

It should be intuitively clear that if programs `P1`, `P2`, `Q1` and `Q2` do not write to variables `v` and `w`, the programs `P1` and `Q2` are never executed at the same time. Similarly for `P2` and `Q1`. A proof of this property for Verilog programs is given in [9] using Duration Calculus. However, we can obtain this result more easily by using theorem 1.

Using the semantics given in this paper we obtain a FLTS for the following program:

```
initial                                  initial
  begin                                    begin
    v=0;                                     w=0;
    inP1=1; chaos(a,b,c); inP1=0;            inQ1=1; chaos(a,b,c); inQ1=0;
    v=1; wait(w);                            w=1; wait(v);
    inP2=1; chaos(a,b,c); inP2=0;            inQ2=1; chaos(a,b,c); inQ2=0;
  end;                                     end;
```

The FLTS is encoded in SMV using a translator we have written (recall that the semantics of a FLTS are independent of the layer information), through which we check that it satisfies the CTL safety property: $AG(\neg(\texttt{inQ1} \wedge \texttt{inP2}) \wedge \neg(\texttt{inP1} \wedge \texttt{inQ2}))$ — "In every reachable state, `inQ1` and `inP2` are mutually exclusive. Similarly for `inP1` and `inQ2`"[2]. The desired result then follows for programs which use no variables other than `a`, `b` and `c` from theorem 1.

This is not more than a toy example. It is clear that for the actual result we desire, we need a stronger theorem — namely that the programs we replace `chaos(`V`)` by, may also use variables which are not used in the program context.

[2] One may object that we have also added the extra assignments in the program, however one can obtain propositions for $inQ1$, $inQ2$, etc in terms of *pos* in the states of the FLTS. Theorem 1 would then also need to be strengthened to allow reasoning about position variables. In our tool, blocks of code can be named so as to automatically produce SMV macros for such properties.

This example only serves to demonstrate how `chaos(`V`)` can be used to provide more than a simply unconstrained global environment.

Our Verilog–to–SMV translator can handle a much larger subset of Verilog than VeriSmall. In particular it handles non-blocking assignments and edge guards (eg `@(posedge v)`) which allow more interesting examples to be constructed. Some other examples specified and verified include counters, simple arithmetic circuits and small algorithms like the one shown above.

9 Related Work

A number of model checkers come together with an abstract language in which transition systems can be specified. Thus, for example, the SMV [8] input language provides a number of high level mechanisms which can be used to specify transition systems. However, while the language allows means of describing complex transition relations, it is rather limited when it comes to means by which transition systems can be combined together. Similarly, Verus [2] provides a high level language in which transition systems can be specified. However, due to the high level nature of the language, one is then left unsure as to whether the semantics specified match those of a Verilog simulator precisely. Since we also view our language semantics specification as a documentation of the semantics, this is undesirable. Another problem is that the the priority levels inherent in Verilog would have to be encoded within the language, introducing another possible source of errors.

The concept of layers corresponds very closely to the idea of priority in process algebra [5]. Usually, however, priorities are associated to transitions or particular language operators, as opposed to particular states. It would be useful to compare our approach to these alternative ones.

The semantics of Verilog have been expressed in terms of a number of variants of transition systems. It is important to note that Verilog has two different semantic interpretations: simulation semantics (which we deal with) and synthesis semantics (which is used in tools which synthesise Verilog code into hardware). Fiskio-Lasserer *et al* [6] express the simulation semantics in terms of an operational semantics while Sasaki [10] has expressed the semantics in terms of abstract state machines. Both provide an excellent documentation of the semantics of the language but do not seem to be particularly suited for proofs about large programs. Gordon *et al* [7] gives the synthesis semantics of the language in terms of transition systems, and the end result of the interpretation is very similar to the one we present. However, the semantics are expressed in terms of a rather complex compilation process which would be rather difficult to prove that it is semantic preserving with respect to other published semantics. The same problem can be found in [3], where a compilation procedure is given to translate programs into finite state machines.

10 Conclusions

We have presented a set of combinators for enriched transition systems. The most important features of our approach are the compositionality and the abstraction which allowed us to express the semantics of VeriSmall so easily. Also, the full semantics of Verilog are just a scaled up version of the semantics of VeriSmall we give here, which is encouraging when one considers the intricate semantics the language has. This work offers us a myriad of opportunities to explore. One of the priorities is the derivation of a number of laws which allow a guaranteed correct implementation of the combinators used. We have implemented a Verilog–to–transition–system translator based on these semantics, which is available upon request from the author. The translator supports a substantially larger subset of Verilog than the one presented in this paper, including non-blocking assignments and guards.

It is generally accepted that any realistic verification of Verilog or VHDL semantics can only be effectively performed at the synthesis level. It is however the case, that simulation is used extensively, and synthesis semantics are different from the related simulation semantics. We are not advocating the verification of large designs at simulation level, but attempt to provide a framework in which the simulation semantics of languages like VHDL and Verilog can be formally reasoned about.

As can be seen from the main theorem in this paper, certain problem solving techniques seem to recur in different languages. The use of a `chaos` constructor, for example, seems to be applicable to most languages. Furthermore, the proof of correctness of the theorem corresponding to the one we give would, in most cases follow the exact same steps. We hope this also to be the case with other results, especially ones related to the generation of a compiler from the source language to transition systems from a given semantics.

Acknowledgements: Thanks to Koen Claessen and Mary Sheeran for their invaluable comments. Thanks also to the anonymous referees for their helpful comments about the paper and how to improve it. Finally, thanks also must go to Walid Taha without whom the language formalised in this paper would not have been 'VeriSmall' but 'a subset of Verilog'.

References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS00*, 2000.
2. S. Campos, E. Clarke, and M. Minea. The Verus tool: A quantitative approach to the formal verification of real-time systems. *Lecture Notes in Computer Science*, 1254, 1997.
3. S. Cheng, R. Brayton, R. York, K. Yelick, and A. Saldanha. Compiling Verilog into timed finite state machines. In *1995 IEEE International Verilog Conference (Washington 1995)*, pages 32–39. IEEE Press, 1995.

4. E. M. Clarke. Automatic verification of finite-state concurrent systems. *Lecture Notes in Computer Science*, 815, 1994.
5. R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000. To appear.
6. John Fiskio-Lasseter and Amr Sabry. Putting operational techniques to the test: A syntactic theory for behavioural Verilog. In *The Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, 1999.
7. Mike Gordon, Thomas Kropf, and Dirk Hoffman. Semantics of the intemediate language IL. Technical Report D2.1c, PROSPER, 1999. available from `http://www.cl.cam.ac.uk` in `/users/mjcg/IL/IL15.ps`.
8. K. McMillan. *Symbolic model checking: An approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
9. Gordon J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In *Proceedings of the Workshop on Formal Techniques in Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.
10. H. Sasaki. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. In *Proceedings of DATE'99 (Design, Automation and Test in Europe), ICM Munich, Germany*, March 1999.
11. IEEE Computer Society. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Computer Society Press, Piscataway, USA, 1996.
12. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *CAV00*, 2000.