From Operational Semantics to Denotational Semantics for Verilog

Zhu Huibiao[1], Jonathan P. Bowen[1], and He Jifeng[2]

[1] Centre for Applied Formal Methods
South Bank University, SCISM, Borough Road, London SE1 0AA, UK
{huibiaz,bowenjp}@sbu.ac.uk http://www.cafm.sbu.ac.uk/
[2] United Nations University, UNU/IIST, P.O. Box 3058, Macau, China
jifeng@iist.unu.edu http://www.iist.unu.edu/

Abstract. This paper presents the derivation of a denotational semantics from an operational semantics for a subset of the widely used hardware description language Verilog. Our aim is to build an equivalence between the operational and denotational semantics. We propose a discrete time semantic model for Verilog. Algebraic laws are also investigated in this paper, with the ultimate aim of providing a unified set of semantic views for Verilog.

1 Introduction

Modern hardware design typically uses a hardware description language (HDL) to express designs at various levels of abstraction. An HDL is a high level programming language, with usual programming constructs such as assignments, conditionals and iterations and appropriate extensions for real-time, concurrency and data structures suitable for modelling hardware. Verilog is an HDL that has been standardized and widely used in industry [6]. Verilog programs can exhibit a rich variety of behaviours, including event-driven computation and shared-variable concurrency.

The semantics for Verilog is very important. At UNU/IIST, the operational semantics has been explored in [1,3,4,7]. Verilog's denotational semantics [9] has also been explored based on the operational semantics using Duration Calculus [8]. The two semantics can be considered equivalent informally. The question is how the two semantics can be proved equivalent formally.

The aim of this paper is to derive the denotational semantics for Verilog from its operational semantics. This ensures the consistency of the two semantics, making it possible to demonstrate their equivalence formally. The similar problem was also investigated in [5] for Dijkstra's sequential language. In our paper we define a transitional condition and the phase semantics for each type of transition. The denotational semantics can be treated as the sequential composition of those phase semantics.

This paper is organized as follows. Section 2 introduces the language and presents a discrete denotational semantic model. We also design a refinement calculus for the discrete model. Section 3 is devoted to deriving the denotational semantics from its operational semantics. We introduce the operational semantics, and define a function that maps any program text to a logic formula representing its denotational semantics.

© Springer-Verlag Berlin Heidelberg 2001

We derive the denotational semantics for each statement from the function by a formal proof in Sect. 4. We also discuss the algebraic laws that are well suited for symbolic calculation. The three semantics form a unifying model, proving different views useful for varying purposes when reasoning about Verilog.

2 The Discrete Denotational Model

2.1 The Syntax for Verilog

The language discussed in this paper is a subset of Verilog. It contains the following categories of syntactic elements introduced in [2].

1. Sequential Process (Thread):
$S ::= PC \mid S\ ;\ S \mid \mathbf{if}\ b\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mid c\ S$
where PC ranges over primitive commands.
$PC ::= (x := e) \mid \mathbf{skip} \mid \mathbf{chaos}$
and $c\ S$ denotes timing controlled statement, and c is a time control used for scheduling.
$c ::= \#(\varDelta) \mid @(\eta)$, where $\eta ::= v \mid\uparrow v \mid\downarrow v$

Time delay $\#\varDelta$ suspends the execution for exactly $\varDelta$ time units. $\varDelta$ is treated as the integer in this paper. Event guard $@(\uparrow v)$ is fired by the increase of the value of v, whereas $@(\downarrow v)$ is triggered by a decrease in v. Any change of v awakes the guard $@(v)$.

2. Parallel Process (Module):
$P ::= S \mid P \parallel P$

To accommodate the expansion laws of parallel construct, the language is equipped with a hybrid control event hc:
$hc ::= @(x := e) \mid @(g) \mid \#(\varDelta)$
$g ::= \eta \mid g\ or\ g \mid g\ and\ g \mid g\ and\ \neg g$
and the guarded choice $(hc_1\ P_1) [\!] \ldots [\!] (hc_n\ P_n)$

2.2 Denotational Semantic Model

Verilog processes are allowed to share program variables. In order to deal with this shared-variable feature, we describe the behaviour of a process in terms of a trace of *snapshots*, which records the sequence of atomic actions in which that process has engaged to some moment in time. Our semantic model contains a variable tr to denote that trace.

If a trace tr is not empty, the function "$last$" yields its last snapshot. Let tr_1, tr_2 be two traces. The notation $tr_1{}^\frown tr_2$ denotes the concatenation of tr_1 and tr_2. $tr_1 \preceq tr_2$ indicates that tr_1 is a prefix of tr_2. Suppose $tr_1 \preceq tr_2$, the notation $tr_2 - tr_1$ denotes the result of subtracting those snapshots in tr_1 from tr_2. We use the notation tr_1 **in** tr_2 to indicate that tr_1 is contained in tr_2, i.e., there are sequences s and t such that $tr_2 = s^\frown tr_1{}^\frown t$.

A snapshot is used to specify the behaviour of an atomic action, and expressed by a triple (t, σ, μ) where:
(1) t indicates the time when the atomic action happens;

(2) σ denotes the final values of program variables at the termination of an atomic action; (3) μ is the control flag indicating which process is in control: $\mu = 1$ states the atomic action is engaged by the process, whereas $\mu = 0$ implies it is performed by the environment.

We select the components of a snapshot using the projections:

$\pi_1((t, \sigma, \mu)) =_{df} t \qquad \pi_2((t, \sigma, \mu)) =_{df} \sigma \qquad \pi_3((t, \sigma, \mu)) =_{df} \mu$

Once a Verilog process is activated, it continues its execution until the completion of an atomic action; namely either it encounters a timing controlled statement, or it terminates successfully. An atomic action usually consists of a sequence of assignments as shown below.

Example 2.1: Consider the parallel program $P \| Q$ where $P =_{df} (x := 1; y := x + 1; z := x + 2)$ and $Q =_{df} x := 2$. Three assignments of P form an atomic action, and their execution is uninterrupted. The process Q can only be started at the beginning or at the end of the execution of P. □

The execution of an atomic action is represented by a single snapshot. In order to describe the behaviour of individual assignment, we introduce a variable ttr to model the accumulated change made by the statements of the atomic action. On the completion of an atomic action, the corresponding snapshot is attached to the end of the trace to record its behaviour.

Example 2.2: Let $P =_{df} x := x + 1 \ ; \ y := y - 1 \ ; \ @(g)$. Assume that program variables x and y are 0 and 1 respectively when P is activated, and the activated time of P is at 0. The execution of $x := x + 1$ produces $ttr = \{x \mapsto 1, \, y \mapsto 1\}$ on its termination that specifies the change made by the assignment to variable x. The statement $y := y - 1$ in turn yields $ttr = \{x \mapsto 1, \, y \mapsto 0\}$ as the final value of ttr, which reflects the change incurred by the atomic action $(x := x+1; y := y-1)$. The snapshot $(0, \, \{x \mapsto 1, \, y \mapsto 0\}, \, 1)$ will be added to the end of the trace variable tr when $@(g)$ is encountered. After this adding, ttr will be assigned an empty value $null$. □

Example 2.3: Let $P =_{df} x := 1 \ ; \ @(x := 2) \ ; \ x := 3$. The contribution of $(x := 1)$ is added to the end of the trace when assignment guard $@(x := 2)$ is encountered. This means $x := 1$ in this particular case is an atomic action. Although $@(x := 2)$ is an atomic action, it also stores its result in ttr. In order to distinguish assignment guard from assignment, we assign a control $flag$ with 0 to identify this case. The result of the assignment guard will be added when its sequential statement is encountered (not only guards). □

We are now ready to represent the observation by a tuple

$$(\overleftarrow{time}, \, \overrightarrow{time}, \, \overleftarrow{tr}, \, \overrightarrow{tr}, \, ttr, \, ttr', \, flag, \, flag')$$

where

• $\overleftarrow{time}$ and $\overrightarrow{time}$ are the start point and the end point of a time interval over which the observation is recorded. We use $\delta(time)$ to represent the length of the time interval.

$$\delta(time) =_{df} (\overrightarrow{time} - \overleftarrow{time})$$

• $\overleftarrow{tr}$ stands for the initial trace of a program over the interval which is passed by its predecessor. $\overrightarrow{tr}$ stands for the final trace of a program over the interval.
$\overrightarrow{tr} - \overleftarrow{tr}$ stands for the sequence of snapshots contributed by the program itself and its

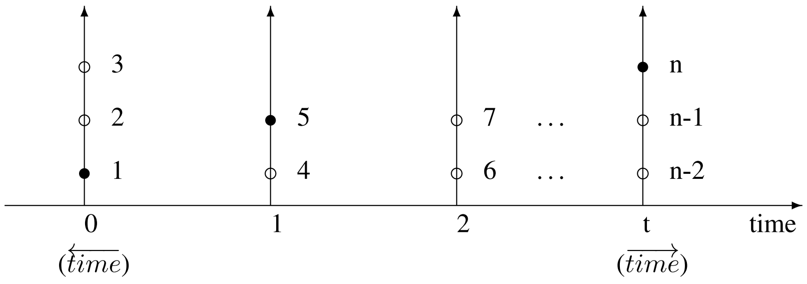
environment during the interval.
• ttr and ttr' stand for the initial and final value of the variable ttr which are used to store the contribution of an atomic action over the interval.
• $flag$ and $flag'$ stand for the initial and final value of the control flag. There are two cases to indicate the end of its prior atomic action("$ttr = null$" or "$ttr \neq null \wedge flag = 0$").

Example 2.4: Let $P =_{df} x := 1 \ ; \ \#1, \quad Q =_{df} \#1 \ ; \ x := 2, \quad R =_{df} x := 3.$ Consider the trace of program $(P \parallel Q) \ ; \ R$.
The trace of P is $< (\overleftarrow{time}, \sigma_1, 1) \ , \ (\overleftarrow{time} + 1, \sigma_2, 0) >$.
The trace of Q is $< (\overleftarrow{time}, \sigma_1, 0) \ , \ (\overleftarrow{time} + 1, \sigma_2, 1) >$.
Hence, the trace of $P \parallel Q$ is $< (\overleftarrow{time}, \sigma_1, 1) \ , \ (\overleftarrow{time} + 1, \sigma_2, 1) >$.
R's trace is $< (\overleftarrow{time}, \sigma_3, 1) >$. Then the trace of $(P \parallel Q) \ ; \ R$ is

$$< (\overleftarrow{time}, \sigma_1, 1) \ , \ (\overleftarrow{time} + 1, \sigma_2, 1) \ , \ (\overleftarrow{time} + 1, \sigma_3, 1) >.$$

where $\sigma_1 = \{x \mapsto 1\}, \quad \sigma_2 = \{x \mapsto 2\}, \quad \sigma_3 = \{x \mapsto 3\}$. □

We use the following diagram to indicate the trace behaviour of a process (and its environment). Here, "•" stands for the process's atomic action. "∘" stands for the environment's atomic action. The numbers on the vertical line stand for the snapshot sequences in the process's trace, whereas the number on the horizontal line represents the time when the atomic actions take place.



As in Temporal Logic, we introduce a binary "chop" operator to describe the composite behaviour of sequential composition.

Definition 2.5

$$P \frown Q =_{df} \exists t, s, tt, f \bullet \quad P[s/\overrightarrow{tr}, t/\overrightarrow{time}, tt/ttr', f/flag'] \\ \wedge \ Q[s/\overleftarrow{tr}, t/\overleftarrow{time}, tt/ttr, f/flag]$$ □

The "chop" operator is associative, and distributes over disjunction. It has I has its unit and **false** as its zero, where

$$I =_{df} \delta(time) = 0 \wedge \overrightarrow{tr} = \overleftarrow{tr} \wedge ttr' = ttr \wedge flag' = flag.$$

A Verilog process may perform an infinite computation and enter a *divergent state*. To distinguish its chaotic behaviour from the stable ones we introduce the variables $ok, ok' : Bool$ into the semantic model, where $ok = true$ indicates the process has been started, and $ok' = true$ states the process has become *stable*.

A timing controlled statement can not start its execution before its guard is triggered. To distinguish its waiting behaviour from terminating one, we introduce another pair of variables $wait, wait' : Bool$. $wait = true$ indicates that the process starts in an intermediate state, and $wait' = true$ means the process is waiting. The introduction of intermediate waiting state has implications for sequential composition "$P; Q$": if Q is asked to start in a waiting state of P, it leaves the state unchanged, i.e., it satisfies the healthiness condition.

$(H)\ Q\ =\ II \lhd wait \rhd Q,$

where $II\ =_{df}\ \mathbf{true} \vdash (\delta(time) = 0) \wedge (\overrightarrow{tr} = \overleftarrow{tr}) \wedge (\bigwedge_{s \in \{ok, wait, ttr, flag\}} s' = s)$

$$P \lhd Q \rhd R\ =_{df}\ (P \wedge Q) \vee (\neg Q \wedge R)$$

$$P \vdash R\ =_{df}\ (ok \wedge P) \Rightarrow (ok' \wedge R)$$

Definition 2.6: Let P and Q be formulae. Define

$$P\ ;\ Q =_{df} \exists w, o \bullet (\ P[w/wait', o/ok']^\frown Q[w/wait, o/ok]\ )$$

Definition 2.7: A formula is called a *healthy formula* if it has the following form.

$$\mathbf{H}(Q \vdash W \lhd wait' \rhd T)$$

where, $\mathbf{H}(X)\ =\ II \lhd wait \rhd X$

Theorem 2.8: $\mathbf{H}(P)$ satisfies healthiness condition (H).

Theorem 2.9: If D_1, D_2 are healthy formulae, so are $D_1 \vee D_2$, $D_1 \lhd b \rhd D_2$ and $D_1\ ;\ D_2$, where

$$\mathbf{H}(Q_1 \vdash W_1 \lhd wait' \rhd T_1)\ ;\ \mathbf{H}(Q_2 \vdash W_2 \lhd wait' \rhd T_2)$$

$$=\ \mathbf{H}(\neg(\neg Q_1 {}^\frown \mathbf{true}) \wedge \neg(T_1 {}^\frown \neg Q_2) \vdash (W_1 \vee (T_1 {}^\frown W_2)) \lhd wait' \rhd (T_1 {}^\frown T_2))$$

Corollary 2.10: If P is a healthy formula then

(1) $II\ ;\ P\ =\ P$ (2) $\bot\ ;\ P\ =\ \bot$ □

The union and intersection of arbitrary healthy formulae set are also healthy formulae. This implies that healthy formulae form a complete lattice under the implication order, which has a bottom element $\bot\ =_{df}\ \mathbf{H}(\mathbf{false} \vdash \mathbf{true})$ and a top element $\top =_{df}\ \mathbf{H}(\mathbf{true} \vdash \mathbf{false})$.

3 From Operational Semantics to Denotational Semantics

3.1 Transitional Condition and Phase Semantics

In order to derive Verilog's denotational semantics from its operational semantics we define a transitional condition and the phase semantics for each type of transition. The standard way to give an operational semantics is to define a set of transition rules based on configurations, such that any computation of a program can be generated from the transition rules. A configuration usually consists of four components (or five components in some cases):

(1) a program text P representing the rest of the program that remains to be executed;
(2) a data state σ (the second element of a configuration) denoting the initial data state of an atomic action;
(3) another data state σ' (the third element) representing the current data state during the execution of an atomic action ($\sigma' = \emptyset$ represents the previous atomic action ends and the new atomic action has not been scheduled);
(4) a control flag k (the fourth element) indicating which process is selected to execute:

$k = 0$ states the program P is waiting to be executed and its environment may perform triggering action or let time make advance, whereas $k = 1$ implies that P is being executed and neither time advance step nor triggering action can take place;
(5) a thread number i (in some configurations) denoting the i-th thread of process P is being executed (i.e., this thread obtains the control flag).

The relationship between a transition and the variables in the denotational model can be described by the following diagram of an example transition.

$$\begin{tikzcd}
& \pi_2(last(\overleftarrow{tr})) & ttr & flag & & \pi_2(last(\overrightarrow{tr})) & ttr' & flag' \\
<P, & \sigma, \arrow[u] & \sigma', \arrow[u] & 0> \arrow[u] \arrow[r, "<\sigma,\sigma'>", "c"'] & <P', & \sigma', \arrow[u] & \emptyset, \arrow[u] & 0> \arrow[u]
\end{tikzcd}$$

Let $O(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ stands for the observation of ttr and $flag$.

$$O(\alpha_1, \alpha_2, \alpha_3, \alpha_4) =_{df} ttr = \alpha_1 \wedge ttr' = \alpha_2 \wedge flag = \alpha_3 \wedge flag' = \alpha_4$$

We use "$ttr = notnull$" to indicate "$ttr \neq null$".

The transition rules can be grouped into the following types [7]. We define a transitional condition $\mathbf{Cond}_{i,j}$ and its corresponding phase semantics for each type of transition. Our map from operational semantics to denotational semantics is based on the phase semantics. Here, $\mathbf{Cond}_{i,j}$ stands for the transitional condition of the $\mathbf{j}$-th transition of type $\mathbf{T}_i$.

• Instantaneous transition

$\mathbf{T}_1$: The i-th thread of process P can perform an instantaneous action, and P enters the instantaneous section by its i-th thread being activated.

$$< P, \sigma, \emptyset, 0 > \longrightarrow < P, \sigma, \sigma, 1, i >, \qquad i \in \{1, 2\}$$
$$\mathbf{Cond}_{1,1} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(null, \pi_2(last(\overleftarrow{tr})), 0, 1)$$

$$< P, \sigma, \sigma', 1 > \longrightarrow < P, \sigma, \sigma', 1, i >, \qquad i \in \{1, 2\}$$
$$\mathbf{Cond}_{1,2} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(notnull, ttr, 1, 1)$$

$\mathbf{T}_2$: Within the instantaneous section, the i-th thread of the process P performs a transition, and remains in the section or terminates. This transition assigns the successor of P an active status.

$$< P, \sigma_0, \sigma, 1, i > \longrightarrow < P', \sigma_0, \sigma', 1, i >, \qquad i \in \{1, 2\}$$
$$< P, \sigma_0, \sigma, 1, i > \longrightarrow < P', \sigma_0, \sigma', 1 >, \qquad i \in \{1, 2\}$$

For a specific program P, σ' should be of the form $f(\sigma)$. The two transitional conditions are the same.

$$\mathbf{Cond}_{2,1} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(notnull, f(ttr), 1, 1)$$

$\mathbf{T}_3$: Within the instantaneous section, the i-th thread of a process may leave the instantaneous section. If the process is breakable, it can also leave the instantaneous section.

$$< P, \sigma_0, \sigma', 1, i > \longrightarrow < P, \sigma_0, \sigma', 0 >, \qquad i \in \{1, 2\}$$
$$< P, \sigma_0, \sigma', 1 > \longrightarrow < P, \sigma_0, \sigma', 0 >$$

The two transitional conditions are the same.

$\mathbf{Cond}_{3,1} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(notnull, ttr, 1, 0)$

$\mathbf{T}_4$: A transition represents that the program executes an assignment guard (i.e., assignment guard is regarded as an atomic action).

$< P, \sigma, \emptyset, 0 > \longrightarrow < P', \sigma, \sigma', 0 >$

For a specific process P, σ' should be of the form $f(\sigma)$.

$\mathbf{Cond}_{4,1} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(null, f(\pi_2(last(\overleftarrow{tr}))), 0, 0)$

The above four types of transitions have the instantaneous feature. The corresponding phase semantics of each transition can be expressed as $Inst(\mathbf{Cond}_{i,j})$.

where, $\mathbf{Cond}_{i,j}$ can be the above seven transitional conditions.

$Inst(\mathbf{X}) =_{df} \mathbf{H}(\mathbf{true} \vdash \neg wait' \wedge \delta(time) = 0 \wedge \mathbf{X}\ )$

"$\delta(time) = 0$" indicates those transitions consume zero time.

• Triggered transition

$\mathbf{T}_5$: (1) A transition can be triggered by its sequential predecessor. This kind of transition is called the self-triggered transition.

$$< P, \sigma, \sigma', 0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_c < P', \sigma', \emptyset, 0 >$$

Here, c in notation $\overset{<\sigma,\sigma'>}{\longrightarrow}_c$ represents the condition which triggers the transition. It has the form $c(\sigma, \sigma')$ based on a pair of states $< \sigma, \sigma' >$. If there is no this kind of condition, it can be understood as $\mathbf{true}$. If σ and σ' (i.e., $\pi_2(last(\overleftarrow{tr}))$ and ttr) are the same, σ' will not be attached to the end of the trace.

$$\begin{aligned}\mathbf{Cond}_{5,1} =_{df}\ & c(\pi_2(last(\overleftarrow{tr})), ttr) \wedge O(notnull, null, 0, 0) \\ & \wedge (\overrightarrow{tr} = \overleftarrow{tr} \lhd \pi_2(last(\overleftarrow{tr})) = ttr \rhd \overrightarrow{tr} = \overleftarrow{tr}^\frown < (\overleftarrow{time}, ttr, 1) >)\end{aligned}$$

This transition also lasts zero time. Its phase semantics is also $Inst(\mathbf{Cond}_{5,1})$.

(2) A transition can be triggered by its parallel partner.

$$< P, \sigma, \emptyset, 0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_c < P', \sigma', \emptyset, 0 >$$

A process can also record the contribution of its environment's atomic action. But the control flag μ in the snapshot is 0. If σ and σ' are the same, the environment will not attach σ' to the end of the trace. Therefore, the process's trace remains unchanged (i.e., $\overrightarrow{tr} = \overleftarrow{tr}$) in this case.

$$\begin{aligned}\mathbf{Cond}_{5,2} =_{df}\ & O(null, null, 0, 0) \wedge c(\pi_2(last(\overleftarrow{tr})), \pi_2(last(\overrightarrow{tr}))) \\ & \wedge \left(\overrightarrow{tr} = \overleftarrow{tr} \vee \begin{pmatrix} \pi_1(\overrightarrow{tr} - \overleftarrow{tr}) = \overleftarrow{time} \wedge \\ \pi_3(\overrightarrow{tr} - \overleftarrow{tr}) = 0 \end{pmatrix} \right)\end{aligned}$$

Its phase semantics is also $Inst(\mathbf{Cond}_{5,2})$. It means its environment's corresponding atomic action also lasts zero time.

• Time advancing transition

$\mathbf{T}_6$: $< P, \sigma, \emptyset, 0 > \xrightarrow{1} < P', \sigma, \emptyset, 0 >$

$\mathbf{Cond}_{6,1} =_{df} \overrightarrow{tr} = \overleftarrow{tr} \wedge O(null, null, 0, 0)$

If process P can not do any other transitions at the moment, time will advance. We regard the unit of time advancing is 1. During this period, there are no atomic actions contributed by the process P itself and its environment. Hence, time advancing keeps the trace unchanged. Its phase semantics is:

$$\mathbf{H}(\mathbf{true} \vdash \mathbf{Cond}_{6,1} \wedge (\delta(time) < 1 \lhd wait' \rhd \delta(time) = 1)\ )$$

3.2 Map from Operational Semantics to Denotational Semantics

Definition 3.1: A configuration $< P, \sigma, \sigma', 1 >$ (or $< P, \sigma, \emptyset, 0 >$) is called *a divergent state* if P can perform an infinite sequence of instantaneous transitions or self-triggered transitions; i.e., there exists an infinite set$\{D_i \,|\, i \in Nat\}$ of configurations such that $D_0 =< P, \sigma, \sigma', 1 >$ (or $< P, \sigma, \emptyset, 0 >$), and for all i,

• either $D_i \longrightarrow D_{i+1}$

• or $D_i =< P_i, \sigma_i, \sigma_i', 0 >, \quad \sigma_i' \neq \emptyset, \quad D_i \xrightarrow{<\sigma_i, \sigma_i'>}_{c_i} D_{i+1}$

where, Nat is the set containing all non-negative integers. □

Definition 3.2: A computational sequence of program P is an empty sequence or any finite sequence leading P to the other state, that is:

$$D_0 \xrightarrow{\delta_1} D_1 \ \ldots\ldots \xrightarrow{\delta_n} D_n$$

where $D_0 =< P,\ \sigma_0,\ \emptyset,\ 0 >$ or $D_0 =< P,\ \sigma_0,\ \sigma_0',\ 1 >$ or $D_0 =< P,\ \sigma_0,\ \sigma_0',\ 0 >$ and $D_i =< P_i,\ \sigma_i,\ \emptyset,\ 0 >$ or $D_i =< P_i,\ \sigma_i,\ \sigma_i',\ 1 >$ or $D_i =< P_i,\ \sigma_i,\ \sigma_i',\ 0 >$ or $D_i =< P_i,\ \sigma_i,\ \sigma_i',\ 1,\ j >$ $(i = 1, \ldots, n$ and $j \in \{1, 2\})$
and $\xrightarrow{\delta_i}$ $(i = 1, \ldots, n)$ can be an instantaneous transition $(\longrightarrow)$, a triggered transition $(\xrightarrow{<\sigma,\sigma'>}_c)$, or a time advancing transition $(\xrightarrow{1})$. □

If computational sequence seq is not empty, $seq[i]$ is the i-th transition $(D_{i-1} \xrightarrow{\delta_i} D_i)$ of seq.

We write $cp[P]$ representing the set which contains all the computational sequences leading program P to terminating state or divergent state. $cp[P]_{ter}$ and $cp[P]_{div}$ stand for the sets which contain all the sequences leading program P to the terminating and divergent states correspondingly. Therefore, we have $cp[P] = cp[P]_{ter} \cup cp[P]_{div}$.

From the operational semantics we know the initial state of process P can be one of the following states before it is executed.

- $< P,\ \sigma,\ \emptyset,\ 0 >$ (represented as $ttr = null$ in the denotational model).
- $< P,\ \sigma,\ \sigma',\ 1 >$ (represented as $ttr \neq null \wedge flag = 1$).
- $< P,\ \sigma,\ \sigma',\ 0 >$ (represented as $ttr \neq null \wedge flag = 0$).

Example 3.3: Let $P =_{df} x := 1; @(\uparrow y)$. Consider the computational sequences of process P under the state $< P,\ \sigma,\ \sigma',\ 1 >$ (operational semantics in the appendix):

$$
\begin{array}{ll|ll}
seq1: & < P,\, \sigma,\, \sigma',\, 1 > & seq2: & < P,\, \sigma,\, \sigma',\, 1 > \\
\longrightarrow & < P,\, \sigma,\, \sigma',\, 1,\, 1 > & \longrightarrow & < P,\, \sigma,\, \sigma',\, 1,\, 1 > \\
\longrightarrow & < @(\uparrow y),\, \sigma,\, \sigma'[1/x],\, 1 > & \longrightarrow & < @(\uparrow y),\, \sigma,\, \sigma'[1/x],\, 1 > \\
\longrightarrow & < @(\uparrow y),\, \sigma,\, \sigma_1,\, 0 > & \longrightarrow & < @(\uparrow y),\, \sigma,\, \sigma_1,\, 0 > \\
\overset{<\sigma,\sigma_1>}{\longrightarrow}_{\neg c} & < @(\uparrow y),\, \sigma_1,\, \emptyset,\, 0 > & \overset{<\sigma,\sigma_1>}{\longrightarrow}_{\neg c} & < @(\uparrow y),\, \sigma_1,\, \emptyset,\, 0 > \\
 & & \overset{<\sigma_1,\sigma_2>}{\longrightarrow}_{\neg c} & < @(\uparrow y),\, \sigma_2,\, \emptyset,\, 0 > \\
 & & \overset{<\sigma_2,\sigma_3>}{\longrightarrow}_{\neg c} & < @(\uparrow y),\, \sigma_3,\, \emptyset,\, 0 > \\
 & & \overset{<\sigma_3,\sigma_4>}{\longrightarrow}_{c} & < \epsilon,\, \sigma_4,\, \emptyset,\, 0 >
\end{array}
$$

where $c = fire(\uparrow y)$ (definition in Sect. 4.4) which means two consecutive states can trigger this guard. Also, $\sigma_1 = \sigma'[1/x]$. $\sigma'[1/x]$ is the same as σ' except mapping x to 1. Here, we find the computational sequence $seq2$ will lead the program to the terminating state (ϵ). □

Example 3.4: Let $Q =_{df} @(\uparrow y); x := 1;$ **chaos**. Consider the computational sequences of process Q under the state $< Q,\, \sigma,\, \emptyset, 0 >$:

$$
\begin{array}{ll|ll}
seq3: & < Q,\, \sigma,\, \emptyset, 0 > & seq4: & < Q,\, \sigma,\, \emptyset, 0 > \\
\overset{<\sigma,\sigma_1>}{\longrightarrow}_{\neg c} & < Q,\, \sigma_1,\, \emptyset,\, 0 > & \overset{<\sigma,\sigma_1>}{\longrightarrow}_{\neg c} & < Q,\, \sigma_1,\, \emptyset,\, 0 > \\
\overset{<\sigma_1,\sigma_2>}{\longrightarrow}_{\neg c} & < Q,\, \sigma_2,\, \emptyset,\, 0 > & \overset{<\sigma_1,\sigma_2>}{\longrightarrow}_{\neg c} & < Q,\, \sigma_2,\, \emptyset,\, 0 > \\
 & & \overset{<\sigma_2,\sigma_3>}{\longrightarrow}_{c} & < x := 1; \mathbf{chaos},\, \sigma_3,\, \emptyset,\, 0 >
\end{array}
$$

Here $c = fire(\uparrow y)$. **chaos** can perform an infinite sequence of instantaneous transitions under any state $< \mathbf{chaos}, \sigma, \sigma', 1 >$ [7]. If "$x := 1;$ **chaos**" takes control at the state $< x := 1; \mathbf{chaos},\, \sigma_3,\, \emptyset,\, 0 >$, it will execute an infinite sequence of instantaneous transitions. Therefore, $seq4$ is the computational sequence leading the program Q to the divergent state. □

$cp[P]_{ter}(0)$ and $cp[P]_{div}(0)$ stand for the sets leading the program to the terminating and divergent states under $< P,\, \sigma,\, \emptyset,\, 0 >$ respectively. $cp[P]_{ter}(1)$ and $cp[P]_{div}(1)$ are the sets leading the program to the terminating and divergent states under $< P,\, \sigma,\, \sigma'$ $,\, 1 >$ correspondingly. $cp[P]_{ter}(2)$ and $cp[P]_{div}(2)$ stand for the sets leading the program to the terminating and divergent states under $< P,\, \sigma,\, \sigma',\, 0 >$ correspondingly. This means:

$$cp[P]_{ter} = cp[P]_{ter}(0) \cup cp[P]_{ter}(1) \cup cp[P]_{ter}(2) \qquad \text{and}$$
$$cp[P]_{div} = cp[P]_{div}(0) \cup cp[P]_{div}(1) \cup cp[P]_{div}(2).$$

Definition 3.5: Let seq stands for a computational sequence of program P. Suppose $len(seq) = n$, $sem(seq)$ is the semantics of the computational sequence seq which can be defined as:
If $len(seq) = 0$ then $sem(seq) =_{df} II$.
If $len(seq) = 1$ then $sem(seq) =_{df} sem_1$.
Otherwise $sem(seq) =_{df} sem_1\ ;\ \ldots\ ;\ sem_n$.
sem_i is the phase semantics of the i-th transition ($seq[i]$) of the computational sequence seq. □

Example 3.6: Let $P =_{df} x := 1\ ;\ x := 2$. There is only one computational sequence seq of P under $< P,\, \sigma,\, \sigma',\, 1 >$:

$$seq : < P, \sigma, \sigma', 1 > \longrightarrow < P, \sigma, \sigma', 1, 1 > \longrightarrow < x := 2, \sigma, \sigma'[1/x], 1 >$$
$$\longrightarrow < x := 2, \sigma, \sigma'[1/x], 1, 1 > \longrightarrow < \epsilon, \sigma, \sigma'[2/x], 1 >$$

The semantics of computational sequence seq is:

$$\begin{array}{ll} \quad sem(seq) & \{\text{Def of 3.5}\} \\ = sem_1 \; ; \; sem_2 \; ; \; sem_3 \; ; \; sem_4 & \{\text{Phase semantics, Th 2.9}\} \\ = Inst(\overrightarrow{tr} = \overleftarrow{tr} \wedge O(notnull, ttr[2/x], 1, 1)) & \Box \end{array}$$

The denotational semantics of program P can be defined as:

Definition 3.7: (Map from operational to denotational)

$$P =_{df} P[0] \lhd ttr = null \rhd (P[1] \lhd flag = 1 \rhd P[2])$$

where

$$P[i] =_{df} \bigvee_{seq \in cp[P]_{div}(i)} (sem(seq) \; ; \; \bot) \quad \vee \quad \bigvee_{seq \in cp[P]_{ter}(i)} (sem(seq)),$$
$$i = 0, 1, 2$$

Here $P[0]$, $P[1]$ and $P[2]$ stand for the semantics of program P under the states $< P,\, \sigma,\, \emptyset,\, 0 >$, $< P,\, \sigma, \sigma',\, 1 >$ and $< P,\, \sigma, \sigma',\, 0 >$ respectively. □

The following definitions and theorems are useful for calculating the denotational semantics for Verilog statements.

Definition 3.8: $< P,\, \sigma_0,\, \emptyset,\, 0 > \quad (\xrightarrow{e}_c)^i \quad < P,\, \sigma_i,\, \emptyset,\, 0 >$

means there exist i steps environment transitions,

$$< P,\, \sigma_0,\, \emptyset,\, 0 > \xrightarrow{<\sigma_0,\, \sigma_1>}_c < P,\, \sigma_1,\, \emptyset,\, 0 > \ldots \xrightarrow{<\sigma_{k-1},\, \sigma_k>}_c < P,\, \sigma_k,\, \emptyset,\, 0 >$$
$$\ldots \xrightarrow{<\sigma_{i-1},\, \sigma_i>}_c < P,\, \sigma_i,\, \emptyset,\, 0 > \quad \Box$$

Definition 3.9: $L1(i)$ stands for the following computational sequence:

$$< P,\, \sigma_0,\, \emptyset,\, 0 > \quad (\xrightarrow{e})^i \quad < P,\, \sigma_i,\, \emptyset,\, 0 > \quad \Box$$

Theorem 3.10: $\bigvee_{i \geq 0} sem(L1(i)) \; = \; (ttr = null) \wedge (flag = 0) \wedge hold(0)$, where
$hold(n) =_{df} \mathbf{H}(\mathbf{true} \vdash idle \wedge ttr' = ttr \wedge flag' = flag \wedge (\delta < n \lhd wait' \rhd \delta = n)\,)$,
$idle =_{df} \pi_3(\overrightarrow{tr} - \overleftarrow{tr}) \in 0^* \wedge incr(\pi_1(\overrightarrow{tr} - \overleftarrow{tr}))$,
$incr(s) =_{df} \forall < t_1, t_2 > \; \mathbf{in} \; s \bullet (t_2 - t_1) \in Nat$ □

Definition 3.11: $< P,\, \sigma,\, \emptyset,\, 0 > \quad (\xrightarrow{et}_c)^{j_0, \ldots, j_\delta} \quad < P,\, \sigma',\, \emptyset,\, 0 >$

means the following detailed computational sequence:

$$\begin{array}{llll} < P, \sigma, \emptyset, 0 > (\xrightarrow{e}_c)^{j_0} & < P, \sigma_1, \emptyset, 0 > \xrightarrow{1} & < P, \sigma_1, \emptyset, 0 > \\ \qquad \ldots & \ldots \qquad\qquad \ldots & \ldots \\ \qquad (\xrightarrow{e}_c)^{j_{\delta-1}} & < P, \sigma_n, \emptyset, 0 > \xrightarrow{1} & < P, \sigma_n, \emptyset, 0 > \\ \qquad (\xrightarrow{e}_c)^{j_\delta} & < P, \sigma', \emptyset, 0 > & & \Box \end{array}$$

where δ is the interval length ($\overrightarrow{time} - \overleftarrow{time}$).

Definition 3.12: $L2(c, j_0, \ldots, j_\delta)$ stands for the following computational sequence.

$$< P, \sigma, \emptyset, 0 > \quad (\xrightarrow{et}_c)^{j_0,\ldots,j_\delta} \quad < P, \sigma', \emptyset, 0 >$$

Theorem 3.13: $\bigvee sem(L2(c, j_0, \ldots, j_\delta)) =_{df} silence(c)$

where, the disjuction "$\bigvee$" is for all $j_0 \geq 0, \ldots, j_\delta \geq 0$.

$$silence(c) =_{df} \mathbf{H}\left(\mathbf{true} \vdash \begin{pmatrix} idle \wedge O(null, null, 0, 0) \wedge \\ \forall < \sigma_1, \sigma_2 > \ \mathbf{in}\ \pi_2(\overrightarrow{tr} - \overleftarrow{tr}) \bullet c(\sigma_1, \sigma_2) \end{pmatrix}\right)$$

$silence(c)$ means during this period, the environment can do any atomic actions, but can not fire the condition $\neg c$.

4 Deriving the Semantics for Statements of Verilog

In this section we will derive the denotational semantics for the Verilog statements by strict proof. Therefore our denotational semantics is equivalent with its operational semantics.

The main purpose of the mathematical definition of Verilog operators is to deduce their interesting properties. These are most elegantly expressed as algebraic laws (equations usually). As our denotational map is based on the transition system of a program, we have two ways to prove the algebraic laws, one using the denotational semantics and the other using the transition system.

4.1 Sequential Composition

The notation $(P \,;\, Q)$ represents the process which behaves like P before P terminates, and then behaves like Q afterwards.

Theorem 4.1: $(P \,;\, Q) \ = \ (P) \,;\, (Q)$

The ";" in the left side is the sequential composition of programs, whereas ";" in the right side is the semantic sequential composition of logic formulae. This theorem indicates the denotational semantics of program $P; Q$ is the sequential composition of their denotational semantics.

4.2 Skip

The role of **skip** is the same as $x := x$ (see operational semantics in the appendix).

Theorem 4.2: $\mathbf{skip} \ = \ flash \lhd (ttr \neq null \wedge flag = 0) \rhd II$
$\qquad\qquad ; \ (hold(0) \,;\, init) \lhd ttr = null \rhd II$

where, $init =_{df} Inst(\ \overrightarrow{tr} = \overleftarrow{tr} \wedge O(null, \pi_2(last(\overleftarrow{tr})), 0, 1)\ )$

$$flash =_{df} Inst\begin{pmatrix} ttr' = null \wedge flag' = 0 \wedge (\ \overrightarrow{tr} = \overleftarrow{tr} \lhd (ttr = null \vee \\ \pi_2(last(\overleftarrow{tr})) = ttr) \rhd \overrightarrow{tr} = \overleftarrow{tr}^\frown < (\overleftarrow{time}, ttr, 1) > \ ) \end{pmatrix}$$

4.3 Assignment

The execution of $x := e$ assigns the value of e to x. Assignment $x := e$ can be in either of the three states before its execution: $< x := e, \sigma, \emptyset, 0 >$, $< x := e, \sigma, \sigma', 1 >$ and $< x := e, \sigma, \sigma', 0 >$.

Case 1: If $ttr = null$, the corresponding computational sequence is :

$$\begin{aligned} < x := e, \sigma, \emptyset, 0 > (\xrightarrow{e})^i < x := e, \sigma_i, \emptyset, 0 > &\longrightarrow < x := e, \sigma_i, \sigma_i, 1, 1 > \\ &\longrightarrow < \epsilon, \sigma_i, \sigma_i[e/x], 1 > \end{aligned}$$

The transitional conditions of the last two instantaneous transitions are: $\mathbf{Cond}_{1,1}$ and $\mathbf{Cond}_{2,2}$.
Let $assign(x := e) =_{df} Inst(\overrightarrow{tr} = \overleftarrow{tr} \wedge ttr' = ttr[e/x] \wedge flag' = flag\ )$
By proof, $Inst(\mathbf{Cond}_{1,1}\ ;\ \mathbf{Cond}_{2,2}) = init\ ;\ assign(x, e)$. Using theorem 3.10, the semantics of $x := e$ in this case is: $hold(0)\ ;\ init\ ;\ assign(x, e)$.

Case 2: If $ttr \neq null \wedge flag = 1$, the corresponding computational sequence is

$$< x := e, \sigma, \sigma', 1 > \longrightarrow < x := e, \sigma, \sigma, 1, 1 > \longrightarrow < \epsilon, \sigma, \sigma[e/x], 1 >$$

The semantics of assignment in this case can be proved as $assign(x, e)$.

Case 3: If $ttr \neq null \wedge flag = 0$, the corresponding computational sequence is:

$$\begin{aligned} < x := e, \sigma, \sigma', 0 > \overset{<\sigma,\sigma'>}{\longrightarrow} &< x := e, \sigma', \emptyset, 0 > \quad (\xrightarrow{e})^i < x := e, \sigma_i, \emptyset, 0 > \\ \longrightarrow \quad &< x := e, \sigma_i, \sigma_i, 1, 1 > \longrightarrow \quad < \epsilon, \sigma_i, \sigma_i[e/x], 1 > \end{aligned}$$

The semantics of $x := e$ under this case is: $flash\ ;\ hold(0)\ ;\ init\ ;\ assign(x, e)$.

Using the semantic map and predicate calculus, we obtain the semantics of assignment.

Theorem 4.3: $x := e \ = \ \mathbf{skip}\ ;\ assign(x := e)$

Verilog assignment statements obey the same set of algebraic laws as its counterpart in the conventional programming languages.

4.4 Event Guard

The guard event is denoted by $@(g)$. A primitive guard g can be of the following forms:
- $\uparrow v$ waits for an increase of the value of v.
- $\downarrow v$ waits for a decrease of the value of v.
- v waits for a change of v.

There are also three types of compound guards.
- g_1 *or* g_2 becomes enabled when either g_1 or g_2 is fired.
- g_1 *and* g_2 becomes enabled if both g_1 and g_2 are awaken simultaneously.
- g_1 *and* $\neg g_2$ becomes fired if g_2 remains idle and g_1 is awaken.

We introduce a predicate $fire(g)(\sigma, \sigma')$ to indicate the transition from the state σ to the state σ' can awake the guard $@(g)$.

$$\begin{aligned} &fire(\uparrow v)(\sigma, \sigma') =_{df} \sigma(v) < \sigma'(v), \quad fire(\downarrow v)(\sigma, \sigma') =_{df} \sigma(v) > \sigma'(v) \\ &fire(v)(\sigma, \sigma') =_{df} \sigma(v) \neq \sigma'(v) \\ &fire(g_1 \text{ or } g_2)(\sigma, \sigma') =_{df} fire(g_1)(\sigma, \sigma') \ \vee \ fire(g_2)(\sigma, \sigma') \end{aligned}$$

$$fire(g_1\ and\ g_2)(\sigma,\sigma') =_{df} fire(g_1)(\sigma,\sigma') \wedge fire(g_2)(\sigma,\sigma')$$
$$fire(g_1\ and\ \neg g_2)(\sigma,\sigma') =_{df} fire(g_1)(\sigma,\sigma') \wedge \neg fire(g_2)(\sigma,\sigma')$$

The event guard $@(g)$ can be immediately fired after it is scheduled, it is actually triggered by the execution of its prior atomic action. According to the operational semantics of $@(g)$ (in the appendix), there are two kinds of computational sequences leading to the terminating state.

$$< @(g),\sigma,\sigma',1 > \longrightarrow < @(g),\sigma,\sigma',0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_{fire(g)} < \epsilon,\sigma',\emptyset,0 >$$

$$< @(g),\sigma,\sigma',0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_{fire(g)} < \epsilon,\sigma',\emptyset,0 >$$

Another case is the guard $@(g)$ waits to be fired by the environment. There are three kinds of computational sequences leading to the terminating state.

$$\begin{aligned} < @(g),\sigma,\sigma',1 > \longrightarrow & < @(g),\sigma,\sigma',0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_{\neg c} < @(g),\sigma',\emptyset,0 > \\ & (\overset{et}{\longrightarrow}_{\neg c})^{j_0,\ldots,j_\delta} < @(g),\sigma_n,\emptyset,0 > \overset{<\sigma_n,\sigma_{n+1}>}{\longrightarrow}_{c} < \epsilon,\sigma_{n+1},\emptyset,0 > \end{aligned}$$

$$\begin{aligned} < @(g),\sigma,\sigma',0 > \overset{<\sigma,\sigma'>}{\longrightarrow}_{\neg c} < @(g),\sigma',\emptyset,0 > (\overset{et}{\longrightarrow}_{\neg c})^{j_0,\ldots,j_\delta} & < @(g),\sigma_n,\emptyset,0 > \\ \overset{<\sigma_n,\sigma_{n+1}>}{\longrightarrow}_{c} & < \epsilon,\sigma_{n+1},\emptyset,0 > \end{aligned}$$

$$< @(g),\sigma,\emptyset,0 > (\overset{et}{\longrightarrow}_{\neg c})^{j_0,\ldots,j_\delta} < @(g),\sigma_n,\emptyset,0 > \overset{<\sigma_n,\sigma_{n+1}>}{\longrightarrow}_{c} < \epsilon,\sigma_{n+1},\emptyset,0 >$$

Here $c = fire(g)$. There is a corresponding phase semantics for each type of transition. Using the definition of phase semantics and Theorem 2.9, 3.13, we obtain:

Theorem 4.4: $@(g) \ = \ selftrig(g) \ \vee \ (await(g)\, ;\, trig(g)\ )$
where,

$$selftrig(g) =_{df} \mathbf{H}(\mathbf{true} \vdash ttr \neq null \wedge fire(g)(\pi_2(last(\overleftarrow{tr})), ttr)) \ \wedge II \ ; \ flash$$

$$\begin{aligned} await(g) =_{df} & \ \mathbf{H}(\ \mathbf{true} \vdash (ttr = null \ \vee \ \neg fire(g)(\pi_2(last(\overleftarrow{tr})), ttr)\ ))\ \wedge \ II \\ & ;\ flash\ ;\ silence(\neg fire(g)) \end{aligned}$$

$$trig(g) =_{df} Inst \begin{pmatrix} idle \wedge len(\overrightarrow{tr} - \overleftarrow{tr}) = 1 \wedge O(null, null, 0, 0) \\ \wedge fire(g)(\pi_2(last(\overleftarrow{tr})),\ \pi_2(last(\overrightarrow{tr}))) \end{pmatrix}$$

4.5 Other Statements

chaos represents the worst process. Its behaviour is totally unpredictable. The conditional **if** $b(v)$ **then** P **else** Q behaves the same as the "**then**" branch if b is true when activated, and the same as the "**else**" branch otherwise. The delay event $\#n$ holds the execution for n units. An assignment guard $@(x := e)$ is a special assignment representing an atomic action. It is used in supporting the parallel expansion laws.

Let $\{g_i \mid 1 \le i \le n\}$ be a finite family of event guards, and $\{P_i \mid 1 \le i \le n\}$ a family of Verilog processes. The notation $(@(g_1)\ P_1) \,\|\, \ldots \,\|\, (@(g_n)\ P_n)$ denotes the program which initially waits for one of the guards to be fired, and then behaves the same as the corresponding guarded process. The program $(@(x_1 := e_1)\ P_1) \| \ldots \| (@(x_n := e_n)\ P_n)$ performs one of its alternative , and the choice is made non-deterministically.

In accordance with the semantic map and operational semantics of these statements [7], we obtain the denotational semantics for these statements.

Theorem 4.5

(1) $\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \ = \ \textbf{skip}\,;\, S_1 \lhd b(ttr) \rhd S_2$

(2) $\#n \ = \ flash\,;\, hold(n)$

(3) $@(x := e) \ = \ flash\ ;\ hold(0)\ ;\ trig(@(x := e))$

(4) $(@(x_1 := e_1)\, P_1) \,[\!]\, \ldots \,[\!]\, \{@(x := e_n)\, P_n)$

$\quad = \ \bigvee \{@(x_i := e_i); P_i \mid 1 \le i \le n\}$

(5) $\textbf{chaos} \ = \ (flash \lhd (ttr \neq null \wedge flag = 0) \rhd II)\ ;\ \bot$

(6) $(@(g_1)\ P_1) \,[\!]\, \ldots \,[\!]\, (@(g_n)\ P_n)$

$\quad = \ \bigvee \{(selftrig(g_i)\ \vee\ (await(g); trig(g_i)))\ ;\ P_i \mid 1 \le i \le n\}$

where $trig(@(x := e)) =_{df} Inst(\ \overrightarrow{tr} = \overleftarrow{tr} \wedge O(null, \pi_2(last(\overleftarrow{tr}))[e/x], 0, 0)\ )$
g stands for the compound guard g_1 *or* $\ldots$ *or* g_n.

4.6 Parallel

Although we have not derived the universal formula representing the denotational semantics for a parallel process, we can write down its transition system. Its semantics can be calculated based on its transition steps. Algebraic laws dealing with parallel can also be proved using the denotational map based on its specific transition systems.

Example 4.6: Let $P \ = \ (\#1; x := 2) \parallel (x := 1; \#1)\ ;\ \#1$. Consider the denotational semantics of P.
We can write down the computational sequences leading program P to the terminating state under three cases ($ttr = null, ttr \neq null \wedge flag = 1$ and $ttr \neq null \wedge flag = 0$) based on the parallel transition [7]. The semantics of P can be calculated based on the semantic map and its computational sequences. Therefore, the denotational semantics of P is

$$flash\ ;\ hold(0)\ ;\ Inst(\ S(1)\ )\ ;\ \ hold(1)\ ;\ Inst(\ S(2)\ )\ ;\ hold(1)$$

where, $S(u) =_{df} \overrightarrow{tr} = \overleftarrow{tr}^\frown < (\overleftarrow{time},\ \{x \mapsto u\}, 1) > \ \wedge\ O(null, null, 0, 0)$ □

Theorem 4.7 (Expansion laws)

(par-1) Let $P_i =_{df}\ @(\eta_i)\ Q_i$ for $i = 1,\ 2$. Then

$$P_1 \| P_2 \ = \ \left(\begin{array}{l} \quad (@(\eta_1\ and\ \neg\eta_2)\ (Q_1 \| P_2)) \\ [\!]\ (@(\eta_1\ and\ \ \eta_2)\ (Q_1 \| Q_2)) \\ [\!]\ (@(\eta_2\ and\ \neg\eta_1)\ (P_1 \| Q_2)) \end{array} \right)$$

(par-2) Let $P_i =_{df}\ @(x_i := e_i)\ Q_i$ for $i = 1,\ 2$. Then

$$P_1 \| P_2 \ = \ \left(\begin{array}{l} \quad (@(x_1 := e_1)\ (Q_1 \| P_2)) \\ [\!]\ (@(x_2 := e_2)\ (P_1 \| Q_2)) \end{array} \right)$$

5 Conclusion

The main contribution of our work is to derive the denotational semantics for a subset of Verilog from its operational semantics [7]. Thus, our denotational semantics presented here is equivalent with its operational semantics. We provide a discrete denotational model and design a refinement calculus for it. Our approach for the derivation is new.

We define a transitional condition and the phase semantics for each type of transition. The denotational semantics can be derived as the sequential composition of those phase semantics. Verilog's algebraic laws are also discussed, which can support program transformation and system partitioning for hardware/software co-design. Proofs are undertaken in two ways, one using the denotational semantics and the other using the operational semantics. Thus, the three semantics form a unifying model for (a subset of) Verilog.

For the future, we are continuing to explore unifying theories for Verilog. We wish to extend the scope of the derivation of denotational semantics for Verilog to further constructs in the language such as iteration. The derivation of operational semantics from denotational semantics for Verilog is another interesting topic for study.

References

1. J. P. Bowen, He Jifeng and Xu Qiwen. An Animatable Operational Semantics of the VERILOG Hardware Description Language. *Proc. ICFEM2000: 3rd IEEE International Conference on Formal Engineering Methods*, IEEE Computer Society Press, pp. 199–207, York, UK, September 2000.
2. M. J. C. Gordon. The Semantic Challenge of Verilog HDL. *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 136–145, June 1995.
3. He Jifeng and Xu Qiwen. *An Operational Semantics of a Simulator Algorithm.* Technical Report 204, UNU/IIST, P.O. Box 3058, Macau, 2000.
4. He Jifeng and Zhu Huibiao. Formalising Verilog. *Proc. IEEE International Conference on Electronics, Circuits and Systems*, IEEE Computer Society Press, pp. 412–415, Lebanon, December 2000.
5. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall International Series in Computer Science, 1998.
6. IEEE *Standard Hardware Description Language based on the Verilog® Hardware Description Language*. IEEE Standard 1364-1995, 1995.
7. Li Yongjian and He Jifeng. *Formalising VERILOG: Operational Semantics and Bisimulation.* Technical Report 217, UNU/IIST, P.O. Box 3058, Macau, November 2000.
8. Zhou Chaochen, C. A. R. Hoare and A. P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.
9. Zhu Huibiao and He Jifeng. A Semantics of Verilog using Duration Calculus. *Proc. International Conference on Software: Theory and Practice,* pp. 421–432, Beijing, China, August 2000.
10. Zhu Huibiao, Jonathan Bowen and He Jifeng. *From Operational Semantics to Denotational Semantics for Verilog*. Technical Report SBU-CISM-01-08, South Bank University, London, UK, May 2001.

Appendix

Below are the transition system definitions for the assignment and event guard constructs. Definitions for other commands can be found in [7].

1. Assignment

$\mathbf{T}_1$: $< v = e, \sigma, \emptyset, 0 > \;\longrightarrow\; < v = e, \sigma, \sigma, 1, 1 >$

$< v = e, \sigma, \sigma', 1 > \;\longrightarrow\; < v = e, \sigma, \sigma', 1, 1 >$

$\mathbf{T}_2$: $< v = e, \sigma, \sigma', 1, 1 > \;\longrightarrow\; < \varepsilon, \sigma, \sigma'[e(\sigma')/v], 1 >$

$\mathbf{T}_5$: $< v = e, \sigma, \sigma', 0 > \;\xrightarrow{<\sigma,\sigma'>}\; < v = e, \sigma', \emptyset, 0 >$

$< v = e, \sigma, \emptyset, 0 > \;\xrightarrow{<\sigma,\sigma'>}\; < v = e, \sigma', \emptyset, 0 >$

2. Event Guard

$\mathbf{T}_3$: $< @(\eta), \sigma, \sigma', 1 > \;\longrightarrow\; < @(\eta), \sigma, \sigma', 0 >$

$\mathbf{T}_5$: $< @(\eta), \sigma, \sigma', 0 > \;\xrightarrow{<\sigma,\sigma'>}_{fire(\eta)}\; < \varepsilon, \sigma', \emptyset, 0 >$

$< @(\eta), \sigma, \emptyset, 0 > \;\xrightarrow{<\sigma,\sigma'>}_{fire(\eta)}\; < \varepsilon, \sigma', \emptyset, 0 >$

$< @(\eta), \sigma, \sigma', 0 > \;\xrightarrow{<\sigma,\sigma'>}_{\neg fire(\eta)}\; < @(\eta), \sigma', \emptyset, 0 >$

$< @(\eta), \sigma, \emptyset, 0 > \;\xrightarrow{<\sigma,\sigma'>}_{\neg fire(\eta)}\; < @(\eta), \sigma', \emptyset, 0 >$

$\mathbf{T}_6$: $< @(\eta), \sigma, \emptyset, 0 > \;\xrightarrow{1}\; < @(\eta), \sigma, \emptyset, 0 >$