Mechanical Approach to Linking Operational Semantics and Algebraic Semantics for Verilog using Maude

Huibiao Zhu[1], Peng Liu[1], Jifeng He[1], and Shengchao Qin[2]

[1]Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai 200062, China
Email: {hbzhu,liup,jifeng}@sei.ecnu.edu.cn
[2]School of Computing, University of Teesside
Middlesbrough TS1 3BA, UK
Email: s.qin@tees.ac.uk

Abstract. Verilog is a hardware description language (HDL) that has been standardized and widely used in industry. It contains interesting features such as event-driven computation and shared-variable concurrency. This paper considers how the algebraic semantics links with the operational semantics for Verilog. Our approach is to apply the equational and rewriting logic system Maude in exploring the linking theories. Firstly we present the algebraic semantics for Verilog. We introduce the concept of head normal form and every program is expressed as a guarded choice with location status. Secondly we present the strategy of deriving operational semantics from algebraic semantics. Our mechanical approach using Maude can visually show the head normal form of each program, as well as the execution steps of a program based on the derivation strategy. Finally we also mechanize the derived operational semantics. The results mechanized from the second and third exploration indicate that the transition system of the derived operational semantics is the same as the one based on the derivation strategy.

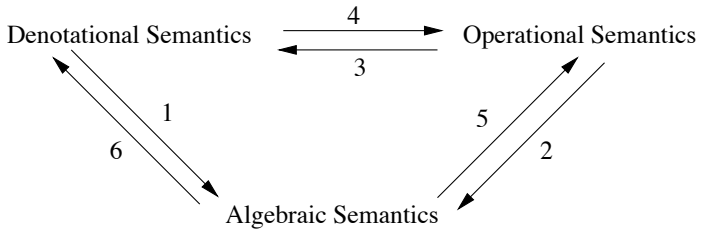
1 Introduction

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 the usual programming constructs such as assignments, conditionals, iterations, together with the 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 [9, 10].

Verilog programs can exhibit a rich variety of behaviours, including event-driven computation, shared-variable concurrency and simulator-based interpretation. Verilog also has real-time features [17], through the time delay statement and the event-driven computation feature.

The semantics for Verilog is very important because it is widely used in industry. The denotational semantics [22] has been investigated using Duration

Calculus [19] in order to describe its real-time features. Various operational semantics have also been studied [5, 6, 11]. Besides the operational and denotational semantics, a set of algebraic laws can also represent the meaning of a language. These three semantics should provide the same understanding of the language from different viewpoints and they should be consistent. Therefore, the linking of these three semantics is a challenging task. Below is the diagram for linking Verilog semantics.



(1) The aim of this step is to generate a set of algebraic laws. These laws can be proved via the achievement of the denotational semantics for Verilog [22].
(2) The aim of this step is also to generate a set of algebraic laws. Compared with step (1), the approach here is based on the operational semantics via bisimulation [15, 16].
(3) Denotational and operational semantics give the meaning for the same language. How can we prove the equivalence and consistency of these two semantics? This step is to derive the denotational semantics from the operational semantics for Verilog.
(4) This aim of this step is to derive the operational semantics from the denotational semantics for Verilog. This gives another way for considering the equivalence and consistency of denotational and operational semantics.
(5) Algebraic semantics also represents the meaning of programs. The aim of this step is to derive the operational semantics from the algebraic semantics.
(6) This step is to derive the denotational semantics back from the algebraic semantics.

Regarding the above linking work of Verilog semantics. Some of them have already been achieved (shown as step (1) to (4), and step (6) in the above diagram). The algebraic laws for Verilog has been verified via the denotational semantics in [22]. These algebraic laws can also be validated based on the operational semantics via bisimulation [6, 11]. Further studies have investigated how the operational semantics relates with denotational semantics for Verilog [20, 21, 23]. We have already investigated the derivation of denotational semantics from operational semantics and algebraic semantics for Verilog respectively [21, 23]. We also derived the operational semantics for Verilog from its denotational semantics [20].

This paper studies how the algebraic semantics for Verilog links with its operational semantics (shown as step (5) in the above diagram). Our approach is

to derive Verilog operational semantics from its algebraic semantics, which can show that the operational semantics is sound and complete with respect to the algebraic laws. We apply the mechanical method to support the semantic linking by using the equational and rewriting logic system Maude [1, 2]. Firstly we present the algebraic semantics for Verilog. We introduce the concept of head normal form and every program is expressed as a guarded choice with location status. In order to investigate the parallel expansion laws, a sequence is introduced, which can indicate an instantaneous action is due to which exact parallel component. Secondly we provide a strategy for deriving operational semantics from algebraic semantics for Verilog. From this strategy, we can achieve a transition system (i.e., an operational semantics). Our mechanical approach using Maude can visually show the head normal form of each program, as well as the execution steps of a program based on the derivation strategy. Finally we also mechanize the derived operational semantics. The results mechanized from the second and third exploration indicate that the transition system of the derived operational semantics is the same as the one based on the derivation strategy.

The remainder of this paper is organized as follows. Section 2 introduces Hardware Description Language Verilog and, as well as Equation and Rewriting Logic system Maude. Section 3 presents a set of algebraic laws, where every program can be represented as a guarded choice with location status. In section 4, we introduce the concept of head normal form and we encode the head normal form of each program in Maude. Section 5 investigates the derivation of the operational semantics from the algebraic semantics. We mechanize the derivation strategy in Maude system. Every program can be executed based on the derivation strategy. For the derived operational semantics, we also explore its mechanical approach. The mechanical approaches from the derivation strategy and the derived operational semantics support the claim that the transition system of the derived operational semantics is the same as the one based on the derivation strategy. Section 6 concludes the paper and provides some future work.

2 Hardware Description Language Verilog and Equational and Rewriting Logic System Maude

2.1 Hardware Description Language Verilog

The Verilog Hardware Description Language (Verilog HDL) became as an IEEE standard in 1995 as IEEE std 1364-1995 [9, 10]. It has many interesting features, such as event-driven computation, shared-variable concurrency and simulator-based interpretation. The syntax of Verilog is expressed in a way that is closer to the syntax of a traditional programming language. Verilog contains the following categories of syntactic elements and is similar to the one introduced by Gordon [3, 4].

$$P ::= PC \mid P \,;\, P \mid \textbf{if } b \textbf{ then } P \textbf{ else } P \mid \textbf{while } b \textbf{ do } P \mid c\, P \mid P \parallel P$$

where:

- PC ranges over primitive commands.
  $PC ::= x := e \mid \mathbf{Skip} \mid @(x := e)$, where
  $x := e$ is the assignment, which is executed exactly once. **Skip** behaves the same as $x := x$. $x := e$ (also **Skip**) is not considered as an atomic action, which is a fragment of an atomic action (i.e., a statement of an atomic action).
  On the other hand, $@(x := e)$ is considered as an atomic action, which is called as atomic assignment.
- $P \ ; \ Q$ is the sequential composition.
- $P \parallel Q$ is the parallel composition, where its mechanism is an interleaving shared-variable concurrency model. The parallel composition can not only be at the outside level, but also can appear at any place.
- $c \ P$ denotes a timing control statement, and c is a time control used for scheduling.

$$\begin{array}{ll} & c ::= \#n \mid @(g) \\ \text{where,} & g ::= \eta \mid g \ or \ g \mid g \ and \ g \mid g \ and \ \neg g \\ & \eta ::= v \mid \uparrow v \mid \downarrow v, \quad n \geq 1 \end{array}$$

  (1) Time delay $\#n$ suspends the execution for exactly n time units, where n is treated as an integer in this paper.

  (2) An 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 $@(v)$.

  (3) $@(g_1 \ or \ g_2)$ becomes enabled if $@(g_1)$ or $@(g_2)$ is fired.

  (4) $@(g_1 \ and \ g_2)$ is triggered if both $@(g_1)$ and $@(g_2)$ are awakened simultaneously.

  (5) $@(g_1 \ and \ \neg g_2)$ becomes fired if $@(g_2)$ remains idle and $@(g_1)$ is awakened.

2.2 Equational and Rewriting Logic System Maude

Rewriting logic has been introduced as a general semantic and logical framework [12–14, 18]. Many applications are implemented in the Maude system [1] and have revealed inspiring results.

In Maude, the fundamental unit can be a functional module or a system module. They can be declared by the following syntax: `fmod NAME is ... endfm` (or `mod NAME is ... endm`) . Here the dots denote the declarations of importing options, sorts, subsorts, operations, equations and rules (only in system modules). First, we take the Peano notation of natural numbers as an example to show the structure of functional modules.

```
fmod PEANO-NATURAL is including BOOL .
```

```
      sorts NzNat Nat .
      subsort NzNat < Nat .
      op 0 : -> Nat [ctor] .
      op s(_) : Nat -> NzNat [ctor] .
      op _+_ : Nat Nat -> Nat .
      vars N M : Nat .
      eq 0 + N = N .
      eq s(M) + N = s(M + N) .
      op _>_ : Nat Nat -> Bool .
      eq s(N) > 0 = true .
      ceq s(N) > s(M) = true if N > M .
      eq N > M = false [owise] .
endfm
```

Defined modules can be reused, as the precluded module `BOOL` is imported into the `PEANO-NATURAL`. Two sorts `NzNat` and `Nat` are declared to represent non-zero natural numbers and natural numbers, and `NzNat` is declared as a `subsort` of `Nat`. `op`s are keywords to define operators on defined sorts. Here, 0 is defined with no operands thus it can be treated as a constant of sort `Nat`. `s(_)` is an operator to define the successor of a natural number, so the result is of sort `NzNat`. We associate the attribute `ctor` (abbreviated for constructor) with these two operators, which means that they are the fundamental operations for defining the canonical forms of the resulting sort. However the operator `+` is not defined as a constructor, because it is not necessary for defining natural numbers. Attributes such as `assoc` and `comm` can also be attached to `op`s, representing that the operator satisfies associative and commutative laws. Variables are declared using the keyword `var(s)` with the sort following behind the name. Equations are defined as simplification rules towards a canonical form. They are declared using the keywords `eq` (i.e., equation) and `ceq` (i.e., conditional equation). We can use the command `red(uce)` to compute the canonical form simplified by equations. When typing in Maude `red s(0) + s(s(0))`, the result will be `s(s(s(0)))`.

In system modules, rewriting rules are declared by keyword `rl` (`crl` for conditional one). Rewriting rules reflect nondeterministic and concurrent transitions of systems. Suppose we define a list of natural numbers as following:

```
mod MY-LIST is including PEANO-NATURAL .
      sorts Elt List .
      subsort Nat < Elt < List .
      op null : -> List [ctor] .
      op _ _ : List List -> List [ctor assoc id: null] .
      vars A B : Elt .
      crl [swap] : A B => B A if A > B .
endm
```

The rule `swap` will make the smaller numbers swap to the left. Any part of the list satisfying this rule will do the transition concurrently. Using the command `rew(rite)` we can see the result of the transitions. `rew s(s(0)) s(s(s(0))) s(0) 0` shows the result as `0 s(0) s(s(0)) s(s(s(0)))`.

As rules in system modules can be nondeterministic and concurrent, rewrite command only shows one of the possible multiple results. Maude provides `search` and `show path` commands to display all possible results. We can see how to use them in later sections.

3 Generating Algebraic Semantics

3.1 Pre-emption Point and Atomic Action

In Verilog, $x := e$, **Skip** and $@(x := e)$ are considered as instantaneous actions. We first introduce the concept of pre-emption point. Only at a pre-emption point does the scheduler make a decision whether the environment gets a chance to make its contribution or the program itself continues to execute.

A pre-emption point can be one of the cases: (1) the two points before and after a timing control statement; (2) the two points before and after a parallel process; (3) for a process that is a component of a parallel process, the two points before and after the process are also pre-emption points. For a sequence of instantaneous actions, if its beginning and ending points are two pre-emption points and there are no pre-emption points appearing inside the sequence, then this sequence is called an atomic action. $x := e$ is not an atomic action. It is a fragment of an atomic action, whereas $@(x := e)$ is an atomic action.

At each pre-emption point for a process, both the process itself and its environment can get the control to do its atomic action. The scheduling between them is non-deterministic. If an instantaneous action is at the beginning of an atomic action, this instantaneous action can be scheduled to execute immediately. Alternatively, the environment can also perform its instantaneous behaviours. After the first instantaneous action in an atomic action terminates, the following actions in the atomic action must be executed sequentially and uninterruptedly. This indicates that the execution of an atomic action is uninterrupted. Regarding the triggering case, a guard can be triggered by its atomic action that has just completed. If there are no cases like this, the guard waits for its environment to trigger it. At some particular points of execution, if the process itself and the environment cannot do any instantaneous action, then time may advance.

Example 3.1

(1) Consider the program $P_1 =_{df} @(\uparrow x)\ ;\ x := 0\ ;\ x := x+1\ ;\ y := x+1\ ;\ \#1$. There are four pre-emption points; i.e., the points before and after the event guard $@(\uparrow x)$ and the two points before and after time delay $\#1$.

(2) Consider the program $P \parallel Q$ when $P =_{df} (x := 0\ ;\ y := x+1\ ;\ z := x+1)$ and $Q =_{df} x := 2$. For process P, there are two pre-emption points; i.e., the point before the assignment $x := 0$ and the point after $z := x+1$. For process Q, the pre-emption points are the points just before and after $x := 2$.

(3) Consider the program $P =_{df} @(\uparrow x)\ ;\ x := 1\ ;\ y := x+1\ ;\ @(x := 0)\ ;\ z := x+1\ ;\ \#1$. The assignment guard $@(x := 0)$ not only assigns a value to x, but also indicates that the previous instantaneous sequence forms an atomic action and itself is an atomic action. This means "$x := 1\ ;\ y := x+1$", "$@(x := 0)$" and "$z := x+1$" are three atomic actions inside program P. □

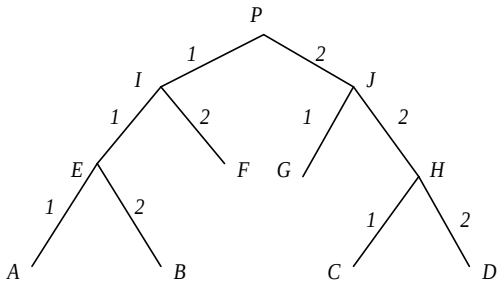
3.2 Locality of Instantaneous Action and Guarded Choice with Location Status

In order to model the scheduling policy for parallel processes, we use a thread sequence *seq* to index the currently active sub-process in a nested parallel com-

position. A thread sequence can be $\langle\rangle$ or a non-empty thread sequence, where $\langle\rangle$ indicates that there is only one thread for the action that has just been executed.

Example 3.2 Let $P =_{df} x := 0 \ ; \ \#1 \ ; \ x := 1 \ ; \ @(\uparrow x)$. The instantaneous actions $x := 0$ and $x := 1$ are both due to process P itself. Therefore, the sequence for indexing their contribution in P is $\langle\rangle$. □

Example 3.3 Let $P = I \parallel J, \ I = E \parallel F, \ J = G \parallel H,$
$E = A \parallel B, \ H = C \parallel D.$
where, the outside structure of processes A, B, F, G, C and D is not the parallel composition. Below is the graph that illustrates the structure of process P.



We assign a label for each edge. If it is the left edge of a process, the label is 1, otherwise the label is 2. Now we consider the sequence that can index the instantaneous action of parallel process P. If P's instantaneous action is due to process A, then the sequence that indexes the exact component of P's contribution is the sequence $\langle 1\rangle\langle 1\rangle\langle 1\rangle$[1]. If P's instantaneous action is due to process B, then the sequence that indexes the exact component of P's contribution is the sequence $\langle 1\rangle\langle 1\rangle\langle 2\rangle$. Similarly, if P's instantaneous action is due to process F, then the sequence that indexes the exact component of P's contribution is the sequence $\langle 2\rangle\langle 2\rangle\langle 2\rangle$. □

Now we introduce the concept of location status for a program, which is one of the following three forms:

(1) *index*, which can be $\langle\rangle$ or a non-empty thread sequence. It indicates an instantaneous action is due to which exact component of a parallel process.

(2) 0, which indicates the termination of an atomic action.

(3) *null*, which indicates a process is at a state where the process itself and its environment can both have a chance to perform its instantaneous action.

For example, in the above example (i.e., Example 3.3), let $A = x := 1 \ ; \ x := x + 1 \ ; \ x := x + 2$. If $x := 1$ is scheduled and completes its execution, the location status for the remaining process of P is $\langle 1\rangle\langle 1\rangle\langle 1\rangle$. This means that it will continue to execute the two subsequent statements $x := x + 1$ and $x := x + 2$.

The introduction of guarded choice is to support the parallel expansion laws.

[1] $\langle 1\rangle\langle 1\rangle\langle 1\rangle$ stands for $\langle 1, 1, 1\rangle$.

Guarded choice can be formalized with location status (i.e., *tag*), as defined below.

Definition 3.3

(1) $h\ (P, tag)$ is a guarded component if it can be one of the forms below. Here, b is a Boolean condition and *index* can be $\langle\rangle$ or a non-empty thread sequence.

$b\&(x := e)\ (P, index), \quad b\&(x := e)\ (P, 0), \quad @(g)\ (P, null), \quad \#1\ (P, null)$

(2) $[\!]\{h_1\ (P_1, tag_1),\ \ldots,\ h_n\ (P_n, tag_n)\}$ is a guarded choice if every element $h_i\ (P_i, tag_i)$ is a guarded component. □

For guarded component "$h\ (P, tag)$", if h is executed (or fired), the subsequent process is P and it is at the location status *tag*. Programs can be represented in the form of a guarded choice. For this aim, the guarded choice can only have five types:

(1) $[\!]_{i\in I}\{b_i\&(x_i := e_i)\ (P_i, tag_i)\}$

(2) $[\!]_{i\in I}\{@(\eta_i)\ (P_i, null)\}$

(3) $[\!]\{\#1\ (P, null)\}$

(4) $[\!]_{i\in I}\{b_i\&(x_i := e_i)\ (P_i, tag_i)\}\ [\!]\ [\!]_{j\in J}\{@(\eta_j)\ (Q_j, null)\}$

(5) $[\!]_{i\in I}\{@(\eta_i)\ (P_i, null)\}\ [\!]\ \{\#1\ (Q, null)\}$

The first type of guarded choice is composed of a set of assignment components, where the second type of guarded choice is only composed of a set of event guard components. The third type is composed of one time-delay component. The fourth type of guarded choice is composed of a set of assignment components and a set of event guard components. The fifth type of guarded choice is composed of a set of event guard components and a time delay component.

When mechanizing in Maude, we implement $b\&(x := e)\ (P, index)$ and $b\&(x := e)\ (P, 0)$ as "`GComponent1`", Similarly, $@(g)\ (P, null)$ and $\#1\ (P, null)$ are implemented as "`GComponent2`" and "`GComponent3`" respectively. All the guarded components are expressed as type "`GComponent`". Below is the detailed definition of guarded components in Maude.

```
fmod GUARDED-COMPONENT is pr VERILOG-PROGRAM .
     pr CONFIG .

     sorts GComponent1 GComponent2 GComponent3 GComponent .
     subsort GComponent1 GComponent2 GComponent3 < GComponent .

     sorts AssignmentGuard GuardPostfix GuardPostfix1 GuardPostfix2
           GuardPostfix3 .
     subsort GuardPostfix1 GuardPostfix2 GuardPostfix3 < GuardPostfix .

     op _&(_) : BoolExp Assignment -> AssignmentGuard [ctor] .
     op `(_,_`) : Program Index -> GuardPostfix1 [ctor] .
     op `(_,_`) : Program EndPoint -> GuardPostfix2 [ctor] .
     op `(_,_`) : Program Null -> GuardPostfix3 [ctor] .

     op __ : AssignmentGuard GuardPostfix1 -> GComponent1 [ctor] .
     op __ : AssignmentGuard GuardPostfix2 -> GComponent1 [ctor] .
     op __ : EventGuard GuardPostfix3 -> GComponent2 [ctor] .
     op __ : TimeControl GuardPostfix3 -> GComponent3 [ctor] .
endfm
```

`GComponent`s are the key components of guarded choices. Based on the three

GComponents, we can define the guarded choices, which we call HealthyGC.

```
subsort HGCType1 HGCType2 HGCType3 HGCType4 HGCType5 < HealthyGC .
op {_} : GComponent1 -> HGCType1 [ctor] .
op {_} : GComponent2 -> HGCType2 [ctor] .
op {_} : GComponent3 -> HGCType3 [ctor] .
op _[]_ : HGCType1 HGCType1 -> HGCType1 [ctor] .
op _[]_ : HGCType2 HGCType2 -> HGCType2 [ctor] .
op _[]_ : HGCType1 HGCType2 -> HGCType4 [ctor] .
op _[]_ : HGCType2 HGCType3 -> HGCType5 [ctor] .
```

In the above definitions, HealthyGC is defined as five subsorts, which are declared as sorts HGCType1, $\cdots$, HGCType5, representing the five types of guarded choices respectively. HGCType1 and HGCType2 are composed of GComponent1 and GComponent2 respectively, separated by []. HGCType3 is composed of a single GComponent3. HGCType4 is concatenated by HGCType1 and HGCType2, and HGCType5 is concatenated by HGCType2 and HGCType3.

3.3 Generating Algebraic Laws

Now we study the expansion laws for parallel composition, which is useful in deriving operational semantics from algebraic semantics. Based on the mechanism of parallel composition for Verilog, we summarize that there are five typical parallel expansion forms, described as *comp*1, *comp*2, $\cdots$, and *comp*5, shown below. The whole set of parallel expansion laws is reflected in the definition of head normal form for parallel composition in the next section. We use the notation $P =_{tag} Q$ to stand for $(P, tag) = (Q, tag)$, indicating that process P and Q are equivalent at location status tag. The notation (P, tag) stands for the behaviour of program P at location status tag.

First we consider the case that one parallel component is in the form of assignment guarded choice. In this case, for a parallel process, no matter which form another parallel component is in, any assignment can be scheduled. The location status of the remaining process after the scheduled assignment should be re-calculated shown in the "**par1**" function[2]. This case can be expressed in "*comp*1", which is defined recursively.

For example, assume $P =_{null} \|_{i\in I}\{b_i \& (x_i := e_i)\ (P_i,\ tag_i)\}$ and

Q be any process.

Then, $\|_{i\in I}\{b_i \& (x_i := e_i)\ \mathbf{par1}(P_i, Q, 1, tag_i))\}$ is one part of the parallel expansion of $P \parallel Q$ due to the initial assignments of P.

Now we consider the case that one parallel component is in the form of event-guarded choice. For a parallel process, we assume that another parallel

---

[2] $\mathbf{par1}(P, Q, i, tag) =_{df} \begin{cases} (\varepsilon,\ 0) & \text{if } P = \varepsilon \text{ and } Q = \varepsilon \\ (\varepsilon \parallel Q,\ 0) & \text{if } P = \varepsilon \text{ and } Q \neq \varepsilon \text{ and } i = 1 \\ (P \parallel \varepsilon,\ 0) & \text{if } P \neq \varepsilon \text{ and } Q = \varepsilon \text{ and } i = 2 \\ (P \parallel Q,\ 0) & \text{if } P \neq \varepsilon \text{ and } tag = 0 \text{ and } i = 1 \\ (P \parallel Q,\ 0) & \text{if } Q \neq \varepsilon \text{ and } tag = 0 \text{ and } i = 2 \\ (P \parallel Q,\ \langle 1\rangle^\frown tag) & \text{if } P \neq \varepsilon \text{ and } tag \neq 0 \text{ and } i = 1 \\ (P \parallel Q,\ \langle 2\rangle^\frown tag) & \text{if } Q \neq \varepsilon \text{ and } tag \neq 0 \text{ and } i = 2 \end{cases}$

component does not have event-guard initially. For this case, if an event guard is fired, the remaining process of the parallel process is the parallel composition of the remaining process of the first component and the second parallel component. This case can be expressed using "*comp*2". Meanwhile, *comp*2 can also be applied to the case that the first parallel component is in the form of time delay component.

For example, assume $Q =_{null} [\!]_{j\in J}\{@(\eta_j)\ (Q_j, null)\}$ and

P does not have event-guard initially.

Then, $[\!]_{j\in J}\{@(\eta_j)\ \mathbf{par}(P, Q_j)\}$[3] is one part of the parallel expansion of $P \parallel Q$ due to the initial event guard firing of Q.

Now we consider the case that both of the two parallel components are in the form of event-guarded choice. There are several types of the triggered cases. If one guard from one parallel part is triggered and all the guards from another parallel part cannot be triggered, the behaviour after the triggered case is the parallel composition of the subsequent process of one parallel part after the triggered guard with another parallel part. This can be defined in "*comp*3" and "*comp*4" recursively. On the other hand, if two guards from different parallel parts are triggered simultaneously, the behaviour after this type of triggered case is the parallel composition of the subsequent processes, after two triggered guards from each parallel part. This can be defined in "*comp*5" recursively.

For example, Assume $P =_{null} [\!]_{i\in I}\{@(\eta_i)\ (P_i, null)\}$ and

$Q =_{null} [\!]_{j\in J}\{@(\xi_j)\ (Q_j, null)\}$

Then

$$[\!]_{i\in I}\{@(\eta_i\ and\ \neg\xi)\ \mathbf{par}(P_i, Q)\} \tag{1}$$

$$\text{and}\quad [\!]_{j\in J}\{@(\xi_j\ and\ \neg\eta)\ \mathbf{par}(P, Q_j)\} \tag{2}$$

$$\text{and}\quad [\!]_{i\in I\wedge j\in J}\{@(\eta_i\ and\ \xi_j)\ \mathbf{par}(P_i, Q_j)\} \tag{3}$$

are the three firing cases for $P \parallel Q$. Here $\eta = or_{i\in I}\{\eta_i\}$ and $\xi = or_{j\in J}\{\xi_i\}$. *comp*3, *comp*4 and *comp*5 stand for the above three firing cases (1), (2) and (3) respectively.

Below is the detailed description of *comp*1, *comp*2, $\cdots$, and *comp*5 in Maude.

```
op comp1(_,_,_) : HGCType1 Program Index -> HGCType1 .
eq comp1({b &(x := e)(P1,tag1)},Q,<1>) = {b &(x := e)par1(P1,Q,<1>,tag1)} .
eq comp1({b &(x := e)(Q1,tag1)},P,<2>) = {b &(x := e)par1(P,Q1,<2>,tag1)} .
eq comp1({h1 Post1} [] hgc',P,i) = comp1({h1 Post1},P,i) []
                                   comp1(hgc',P,i) .
op comp2(_,_,_) : HealthyGC Program Index -> HealthyGC .
eq comp2({@(g)(P1,tag1)},Q,<1>) = {@(g)(par(P1,Q),tag1)} .
eq comp2({@(g)(Q1,tag1)},P,<2>) = {@(g)(par(P,Q1),tag1)} .
eq comp2({# 1(P1,tag1)},Q,<1>) = {# 1(par(P1,Q),tag1)} .
```

[3] $\mathbf{par}(P, Q) =_{df} \begin{cases} (\varepsilon, null) & \text{if} P = \varepsilon \text{ and } Q = \varepsilon \\ (P \parallel Q, null) & \text{otherwise} \end{cases}$

```
eq comp2({# 1(Q1,tag1)},P,<2>) = {# 1(par(P,Q1),tag1)} .
eq comp2({h1 Post1} [] hgc',P,i) = comp2({h1 Post1},P,i) []
                                   comp2(hgc',P,i) .
op comp3(_,_,_) : HGCType2 HGCType2 Program -> HGCType2 .
eq comp3({@(g1)(P1,null)}, hgct2, Q) = {@(g1 and ~ guard(hgct2))
                                        (par(P1,Q),null)} .
eq comp3({@(g1)(P1,null)} [] hgct2', hgct2, Q) =
              comp3({@(g1)(P1,null)},hgct2,Q) [] comp3(hgct2',hgct2,Q) .
op comp4(_,_,_) : HGCType2 HGCType2 Program -> HGCType2 .
eq comp4(hgct2, {@(g1)(Q1,null)}, P) = {@(g1 and ~ guard(hgct2))
                                        (par(P,Q1),null)} .
eq comp4(hgct2, {@(g1)(Q1,null)} [] hgct2', P) =
               comp4(hgct2,{@(g1)(Q1,null)},P) [] comp4(hgct2,hgct2',P) .
op comp5(_,_) : HGCType2 HGCType2 -> HGCType2 .
eq comp5({@(g1)(P1,null)},{@(g2)(Q1,null)}) = {@(g1 and g2)
                                               (par(P1,Q1),null)} .
eq comp5({@(g1)(P1,null)},{@(g2)(Q1,null)} [] hgct2) =
comp5({@(g1)(P1,null)},{@(g2)(Q1,null)}) [] comp5({@(g1)(P1,null)},hgct2) .
eq comp5({@(g1)(P1,null)} [] hgct2,hgct2') =
                  comp5({@(g1)(P1,null)},hgct2') [] comp5(hgct2,hgct2') .
```

All these five definitions are used to compute the components of parallel expansions of two processes. In order to compute the remaining process after the corresponding guard of the parallel expansion, the process not being scheduled will be added as a parameter to the computation.

4 Generating Head Normal Form

In order to support the derivation of operational semantics from algebraic semantics, we introduce the concept of head normal form. For program P, if its location status is tag, we use the notation $HF(P, tag)$ to stand for the head normal form of process P at the location status tag.

The head normal form of $HF(P, tag)$ is to make one step forward expansion for program P at the location status tag. For parallel program P, the parallel expansion laws can help to calculate the head normal form $HF(P, tag)$. As the operational semantics is also to make one step forward transition, the head normal form can support to derive the operational semantics.

4.1 Sequential Constructs

For sequential constructs, its initial location status can be $null$, $\langle\rangle$ and 0. Below are the definitions of the head normal form of sequential constructs at the location status tag.

```
eq   HF(x := e,tag) = ({t &(x := e)(nil,<>)},tag) .
eq   HF(Skip,tag) = ({t &(Skip)(nil,<>)},tag) .
eq   HF(@(x := e),tag) = ({t &(x := e)(nil,0)},tag) .
eq   HF(@(g),tag) = ({@(g)(nil,null)},tag) .
eq   HF(# 1,tag) = ({# 1(nil,null)},tag) .
eq   HF(# n,tag) = ({# 1(# (n - 1),null)},tag) .
ceq  HF(P ; Q,tag) = (seq(T,Q),tag) if (T,tag) := HF(P,tag) .
ceq  HF(P ; Q,tag) = (seq(T,Q),tag) if (T,tag) := HF(P,tag) .
ceq  HF(if b then P else Q,tag) = ({b &(Skip)(P,<>)} [] {~ b &(Skip)(Q,<>)},tag) .
ceq  HF(while b do P,tag) = ({b &(Skip)(P ; while b do P,<>)} []
                             {~ b &(Skip)(nil,<>)},tag) .
```

The first line defines the head normal form of $x := e$. Here "t" stands for $true$ and "nil" stands for the empty process ε. After its execution, the location status is $\langle\rangle$. On the other hand, the third line defines the head normal form of assignment guard $@(x := e)$. After its execution, the location status is 0, indicating the completion of an atomic action.

4.2 Parallel Composition

For parallel process $P \parallel Q$, its location status can be $null$, 0 and seq. Firstly we consider the head normal form of $P \parallel Q$ at the location status $null$.

There are five types of guarded choice. We first consider the case that two parallel components of a parallel process are of the first three types. Their head normal forms are defined based on $comp1$, $comp2$, $\cdots$, $comp5$.

```
ceq    HF(P || Q,null) = (comp1(hgct11,Q,<1>) [] comp1(hgct12,P,<2>) , null)
       if (hgct11,null) := HF(P,null) /\ (hgct12,null) := HF(Q,null) .
ceq    HF(P || Q,null) = (comp1(hgct1,Q,<1>) [] comp2(hgct2,P,<2>) , null)
       if (hgct1,null) := HF(P,null) /\ (hgct2,null) := HF(Q,null) .
ceq    HF(P || Q,null) = (comp1(hgct1,Q,<1>) , null)
       if (hgct1,null) := HF(P,null) /\ (hgct3,null) := HF(Q,null) .
ceq    HF(P || Q,null) = (comp3(hgct21,hgct22,Q) [] comp4(hgct21,hgct22,P) []
                          comp5(hgct21,hgct22), null)
       if (hgct21,null) := HF(P,null) /\ (hgct22,null) := HF(Q,null) .
ceq    HF(P || Q,null) = (comp2(hgct2,Q,<1>) [] comp2(hgct3,P,<2>), null)
       if (hgct2,null) := HF(P,null) /\ (hgct3,null) := HF(Q,null) .
ceq    HF(P || Q,null) = ({# 1(par(R1,R2),null)}, null)
       if ({# 1(R1,null)},null) := HF(P,null) /\ ({# 1(R2,null)},null) := HF(Q,null) .
```

In the above definitions, we use the first case to explain our definition. The head normal forms of P and Q at the location status $null$ are expressed as `hgct11` and `hgct12` respectively, and they are both of the first type of guarded choice. Hence, "`comp1(hgct11,Q,<1>)[]comp1(hgct12,P,<2>)`" is the head normal form of $P \parallel Q$ at the location status $null$, indicating that the assignments in P and Q can both be scheduled first and the location status after the execution of the corresponding assignment is calculated.

If a process is in the form of the fourth type of guarded choice (or the fifth type of guarded choice), it can be composed in parallel with any process (i.e., in the form of any type of guarded choice). The head normal form of the corresponding parallel process at location status $null$ can be defined similarly. Below is the case that both of the parallel components are of the fourth type of guarded choice.

```
ceq    HF(P || Q,null) = (comp1(hgct11,Q,<1>) [] comp1(hgct12,P,<2>) []
                          comp3(hgct21,hgct22,Q) [] comp4(hgct21,hgct22,P) []
                          comp5(hgct21,hgct22),null)
       if (hgct11[]hgct21,null) := HF(P,null) /\ (hgct12[]hgct22,null) := HF(Q,null) .
```

The above analysis considered the generating of head normal form for a parallel process at the location status $null$. Now we consider other cases for the location status of a parallel process. The first four **ceq**s below explore the case that one parallel part is at the state of the execution of an instantaneous action.

```
ceq    HF(P || Q,<1> ^ index) = (comp1({b &(x := e)(P1,index)},Q,<1>), <1> ^ index)
       if ({b &(x := e)(P1,index)},index) := HF(P,index) .

ceq    HF(Q || P,<2> ^ index) = (comp1({b &(x := e)(P1,index)},Q,<2>), <2> ^ index)
       if ({b &(x := e)(P1,index)},index) := HF(P,index) .

ceq    HF(P || Q,<1> ^ index) = (comp1({b &(x := e)(P1,0)},Q,<1>), <1> ^ index)
       if ({b &(x := e)(P1,0)},index) := HF(P,index) .

ceq    HF(Q || P,<2> ^ index) = (comp1({b &(x := e)(P1,0)},Q,<2>), <2> ^ index)
       if ({b &(x := e)(P1,0)},index) := HF(P,index) .

ceq    HF(P || Q,index) = (T,index) if (T,null) := HF(P || Q,null) .

ceq    HF(P || Q,0) = (T,0) if (T,null) := HF(P || Q,null) .
```

In the above definitions, the first and second case models the situation that a process continues to execute the next assignment in an atomic action. For parallel process $P \parallel Q$, the first case models the execution of next assignment which is contributed by process P, whereas the second case models the execution of next assignment which is due to Q. Now we explain the first case further. For process P at location status `index`, after the execution of the next assignment, the location status is still `index`. Then, for parallel process, it will execute the same next assignment contributed by P and the location status is expressed as "`<1>^index`".

Example 4.1 Let $P = ((x := x+1\ ;\ @(\uparrow y)) \parallel y := y+1) \parallel (\#1\ ;\ y := y+1)$. Now we consider the head normal form of process P at the location state *null*.

For process P, two assignments $x := x+1$ and $y := y+1$ in $(x := x+1\ ;\ @(\uparrow y)) \parallel y := y+1$ can have chances to be scheduled. Therefore, if $x := x+1$ is scheduled, the remaining process is $(@(\uparrow y) \parallel y := y+1) \parallel (\#1\ ;\ y := y+1)$ and the corresponding location status is $\langle 1\rangle\langle 1\rangle$.

On the other hand, for process P, if $y := y+1$ in $(x := x+1\ ;\ @(\uparrow y)) \parallel y := y+1$ is scheduled, the remaining process is $(x := x+1\ ;\ @(\uparrow y)) \parallel (\#1\ ;\ y := y+1)$. As this assignment is the last statement of parallel composition $(x := x+1\ ;\ @(\uparrow y)) \parallel y := y+1$, the location status is 0 after the execution of this assignment.

Using the command `red(uce)` provided by Maude, we can compute the head normal form of the example program,

```
reduce in HEAD-NORM-FORM :
   (    {t & (x := x + 1) ((@(↑ y) ‖ y := y + 1) ‖ (#1 ; y := y + 1) , ⟨1⟩⟨1⟩)}
     ‖ {t & (y := y + 1) ((x := x + 1 ; @(↑ y)) ‖ (#1 ; y := y + 1) , 0)}
   , null  )
```

5 Generating Operational Semantics from Algebraic Semantics

5.1 Transition Types

The transitions for Verilog are of the form $C \xrightarrow{\beta} C'$, where C and C' are configurations describing the states of an executing mechanism before and after a step respectively. Here we use β to represent the transition type. There are three types of configurations:

$$\langle P,\, \sigma,\, \emptyset\rangle \qquad \langle P,\, \sigma,\, \sigma',\, 1,\, seq\rangle \qquad \langle P,\, \sigma,\, \sigma',\, 0\rangle$$

where:

(1) The first component P is a program text representing the program that remains to be executed.

(2) The second component σ in $\langle P,\, \sigma,\, \emptyset\rangle$ stands for the initial state, which can be regarded as the initial state of the atomic action that appears at the beginning of P. The second component σ in other configurations stands for the initial state of an atomic action that is currently being executed.

(3) The third component σ' ($\sigma' \neq \emptyset$) models the accumulation of the contribution of instantaneous actions in an atomic action. If the third component is $\emptyset$, it means the previous atomic action ends and the new atomic action has not been scheduled.

(4) If the third component is not empty, a control flag j should be supplied in the configuration as the fourth element. "$j = 0$" indicates that the current atomic action ends, where "$j = 1$" indicates that current atomic action is still executing.

(5) In order to model the scheduling policy for parallel processes, a thread sequence seq is supplied in the configuration if the third element is not empty (for explanations, see section 3.2), which is used to index the currently active sub-process in a nested parallel composition. Here, seq can be $\langle\rangle$ or a non-empty thread sequence.

The transition rules for Verilog programs can be grouped into the following three types: (1) Instantaneous transition $C \longrightarrow C'$; (2) Event transition $C \xrightarrow{\langle\sigma,\sigma'\rangle} C'$; (3) Time advance transition $C \xrightarrow{1} C'$. Below are the detailed descriptions:

1. Instantaneous transition

   $\mathbf{T}_1$ A process can perform its first instantaneous action of an atomic action.
   $$\langle P,\ \sigma,\ \emptyset\rangle \ \longrightarrow\ \langle P',\ \sigma,\ \sigma',\ 1,\ \ seq\rangle$$

   $\mathbf{T}_2$ A process can continue its following instantaneous action in an atomic action.
   $$\langle P,\ \sigma,\ \sigma',\ 1,\ seq\rangle \ \longrightarrow\ \langle P',\ \sigma,\ \sigma'',\ 1,\ seq\rangle$$

   $\mathbf{T}_3$ A process completes an instantaneous section.
   $$\langle P,\ \sigma,\ \sigma',\ 1,\ seq\rangle \ \longrightarrow\ \langle P,\ \sigma,\ \sigma',\ 0\rangle$$

   $\mathbf{T}_4$ A process executes an assignment guard.
   $$\langle P,\ \sigma,\ \emptyset\rangle \ \longrightarrow\ \langle P',\ \sigma,\ \sigma',\ 0\rangle$$

2. Event transition

   $\mathbf{T}_5$ (1) A transition can be fired by the atomic action that has just completed. $\langle P,\ \sigma,\ \sigma',\ 0\rangle \ \xrightarrow{\langle\sigma,\sigma'\rangle}\ \langle P',\ \sigma',\ \emptyset\rangle$

   (2) A transition can be fired by the action of its environment.
   $$\langle P,\ \sigma,\ \emptyset\rangle \ \xrightarrow{\langle\sigma,\sigma'\rangle}\ \langle P',\ \sigma',\ \emptyset\rangle$$

3. Time advance transition

   $\mathbf{T}_6$ A process that cannot do anything else will allow time to advance. Time advances in unit steps.
   $$\langle P,\ \sigma,\ \emptyset\rangle \ \xrightarrow{1}\ \langle P',\ \sigma,\ \emptyset\rangle$$

The configuration can be implemented in Maude as below.

```
fmod CONFIG is pr VERILOG-PROGRAM .
      pr ENVIRONMENT .
      ......
      op # : -> Init [ctor] .
      op 1 : -> Flag [ctor] .
      op 0 : -> EndPoint [ctor] .
      op <> : -> Index [ctor] .
      op <1> : -> Index [ctor] .
      op <2> : -> Index [ctor] .
      op _^_ : Index Index-> Index [ctor assoc id: <>] .

      op <_,_,_> : Program Env Init -> Config [ctor] .
      op <_,_,_,_,_> : Program Env Env Flag Index -> Config [ctor] .
      op <_,_,_,_> : Program Env Env EndPoint -> Config [ctor] .
endfm
```

The definition of configurations in Maude is based on the three types of configurations. The first type contains three components and ends with a `#` which is the only element of sort `Init` (i.e., representing $\emptyset$). And the second type contains five components among which the fourth is the `Flag` represented by `1` and the fifth is a `Index`. The third type contains four components and the fourth is the `EndPoint` represented by `0`.

5.2 Deriving Operational Semantics from Algebraic Semantics

The main purpose of this section is to derive the transition system for Verilog from its algebraic laws. This approach allows the operational semantics to be derived as theorems, rather than being presented as postulates or definitions.

Firstly we give the derivation strategy, which is based on the head normal of each program. For every program, its location status can be *null*, $\langle\rangle$ or *seq*.

Definition 5.1 (Derivation Strategy)

(1.a) If $HF(P, null) = (\ [\!]_{i\in I}\{b_i \& (x_i := e_i)\ (P_i, tag_i)\},\ \ null\ )$,
then P can perform transitions at states $\langle P, \sigma, \emptyset\rangle$ and $\langle P, \sigma, \sigma', 0\rangle$.

```
crl [1.a1] : . < P , env , # > => < Pi , env , env <- (x , e) , 1 , tag >
       if (hgct1,null) := HF(P,null) /\ (hgc [] {b &(x := e)(Pi,tag)} [] hgc', null)
       := (hgct1,null) /\ b[env] /\ tag =/= 0 .
crl [1.a1'] : . < P , env , # > => < Pi , env , env , 1 , tag >
       if (hgct1,null) := HF(P,null) /\ (hgc [] {b &(Skip)(Pi,tag)} [] hgc', null)
       := (hgct1,null) /\ b[env] /\ tag =/= 0 .
crl [1.a2] : . < P , env , # > => < Pi , env , env <- (x , e) , 0 >
       if (hgct1,null) := HF(P,null) /\ (hgc [] {b &(x := e)(Pi,tag)} [] hgc', null)
       := (hgct1,null) /\ b[env] /\ tag == 0 .
crl [1.a3] : . < P , env , env' , 0 > => < P , env' , # >
       if (hgct1,null) := HF(P,null) .
```

(1.b) If $HF(P, null) = (\ [\!]_{i\in I}\{@(\eta_i)\ (P_i, null)\},\ \ null\ )$,
then P can perform transitions at states $\langle P, \sigma, \emptyset\rangle$ and $\langle P, \sigma, \sigma', 0\rangle$.

```
crl [1.b1] : . < P , env , env' , 0 > => < P , env' , # >
       if (hgct2,null) := HF(P,null) /\ not fire(guard(hgct2))(env,env') .
crl [1.b2] : . < P , env , env' , 0 > => < Pi , env' , # >
       if (hgct2,null) := HF(P,null) /\ (hgc [] {@(g)(Pi,null)} [] hgc' , null) :=
       (hgct2,null) /\ fire(g)(env,env') .
crl [1.b3] : . < P , env , # > => < P , env , # >
       if (hgct2,null) := HF(P,null) .
```

(1.c) If $HF(P, null) = (\ [\!]\{\#1\ (R, null)\},\ \ null)$,
then P can perform transitions at states $\langle P, \sigma, \emptyset\rangle$ and $\langle P, \sigma, \sigma', 0\rangle$.

```
crl [1.c1] : < P , env , env' , 0 > => < P , env' , # >
       ({# 1(R,null)} , null) := HF(P,null) .

crl [1.c2] : < P , env , # > => < R , env , # >
       ({# 1(R,null)} , null) := HF(P,null) .
```

(1.d) If $HF(P, null) \;=\; (\; \|_{i\in I}\{b_i\&(x_i := e_i)\ (P_i,\, tag_i)\} \parallel \|_{j\in J}\{@(\eta_j)\ (R_j, null)\},\; null\;)$,
then P can perform transitions at states $\langle P, \sigma, \emptyset\rangle$ and $\langle P, \sigma, \sigma', 0\rangle$.

```
crl [1.d1] : . < P , env , # > => < Pi , env , env <- (x , e) , 1 , tag >
       if (hgct1 [] hgct2,null) := HF(P,null) /\ (hgc [] {b &(x := e)(Pi,tag)} [] hgc',
       null):= (hgct1,null) /\ b[env] /\ tag =/= 0 .

crl [1.d1'] : . < P , env , # > => < Pi , env , env , 1 , tag >
       if (hgct1 [] hgct2,null) := HF(P,null) /\ (hgc [] {b &(Skip)(Pi,tag)} [] hgc',
       null) := (hgct1,null) /\ b[env] /\ tag =/= 0 .

crl [1.d2] : . < P , env , # > => < Pi , env , env <- (x , e) , 0 >
       if (hgct1 [] hgct2,null) := HF(P,null) /\ (hgc [] {b &(x := e)(Pi,tag)} [] hgc',
       null) := (hgct1,null) /\ b[env] /\ tag == 0 .

crl [1.d3] : . < P , env , env' , 0 > => < P , env' , # >
       if (hgct1 [] hgct2,null) := HF(P,null) /\ not fire(guard(hgct2))(env,env') .

crl [1.d4] : . < P , env , env' , 0 > => < R , env' , # >
       if (hgct1 [] hgct2,null) := HF(P,null) /\ (hgc [] {@(g)(R,null)} [] hgc' , null)
       := (hgct2,null) /\ fire(g)(env,env') .
```

(1.e) If $HF(P,\, null) \;=\; (\; \|_{i\in I}\{@(\eta_i)\ (P_i,\, null)\} \parallel \{\#1\ (R,\, null)\},\;\; null\;)$,
then P can perform transitions at states $\langle P, \sigma, \emptyset\rangle$ and $\langle P, \sigma, \sigma', 0\rangle$.

```
crl [1.e1] : . < P , env , env' , 0 > => < P , env' , # >
    if (hgct2 [] {# 1(R,null)},null) := HF(P,null) /\ not fire(guard(hgct2))(env,env') .

crl [1.e2] : . < P , env , env' , 0 > => < R , env' , # >
    if (hgct2 [] {# 1(R,null)},null) := HF(P,null) /\ (hgc [] {@(g)(R,null)} [] hgc'
    , null) := (hgct2,null) /\ fire(g)(env,env') .

crl [1.e3] : . < P , env , # > => < R , env' , # >
    if (hgct2 [] {# 1(R,null)},null) := HF(P,null) .
```

(2.a) If $HF(P,\, seq) \;=\; (\; \|_{i\in I}\{b\&(x_i := e_i)\ (P_i,\, seq)\},\;\; seq\;)$,
then P can perform transitions at state $\langle P, \sigma, \sigma', 1, seq\rangle$.

```
crl [2.a] : . < P , env , env' , 1 , index > => < Pi , env , env <- (x , e) , 1 , index >
       if (hgct1,index) := HF(P,index) /\ (hgc [] {b &(x := e)(Pi,index)} [] hgc',
       index) := (hgct1,index) /\ b[env] .

crl [2.a'] : . < P , env , env' , 1 , index > => < Pi , env , env' , 1 , index >
       if (hgct1,index) := HF(P,index) /\ (hgc [] {b &(Skip)(Pi,index)} [] hgc',
       index) := (hgct1,index) /\ b[env] .
```

(2.b) If $HF(P,\, seq) \;=\; (\; \|_{i\in I}\{b_i\&(x_i := e_i)\ (P_i,\, 0)\},\;\; seq\;)$,
then P can perform transitions at state $\langle P, \sigma, \sigma', 1, seq\rangle$.

```
crl [2.b] : . < P , env , env' , 1 , index > => < Pi , env , env <- (x , e) , 0 >
       if (hgct1,index) := HF(P,index) /\ (hgc [] {b &(x := e)(Pi,0)} [] hgc',
       index) := (hgct1,index) /\ b[env] .

crl [2.b'] : . < P , env , env' , 1 , index > => < Pi , env , env' , 0>
       if (hgct1,index) := HF(P,index) /\ (hgc [] {b &(Skip)(Pi,0)} [] hgc',
       index) := (hgct1,index) /\ b[env] .

crl [2.b"] : . < P , env , env' , 1 , index > => < P , env , env' , 0 >
       if (hgct1 [] hgct2,index) := HF(P,index) .

crl [2.b"'] : . < P , env , env' , 1 , index > => < P , env , env' , 0 >
       if (hgct2 [] {# 1(R,null)},index) := HF(P,index) .
```

(3.a) If $HF(P,\, \langle\rangle) \;=\; (\; \|_{i\in I}\{g_i\ (P_i,\, tag_i)\},\;\; \langle\rangle\;)$ and $\forall i \in I \bullet tag_i \neq \langle\rangle$,

then P can perform transitions at state $\langle P, \sigma, \sigma', 1, \langle\rangle\rangle$.

```
crl [3.a] : . < P , env , env' , 1 , <> > => < P , env , env' , 0 >
      if (hgc , <>) := HF(P , <>) /\ tagnotempty(hgc) .
```

For the above derivation strategy, items ((1.a)–(1.e)) explore the situation that a program is at the location status *null*. The corresponding derivation strategy can be defined based on the five types of guarded choice of the head normal form of a program. If the head normal form of a program is expressed as the first type of guarded choice, the program can perform the first instantaneous action of an atomic action provided that the location status of the subsequent process is not 0. On the other hand, if the location status of the subsequent process is 0, this means that the program can perform an assignment guard transition. Meanwhile, the program can also perform a transition of event transition type. This can be expressed in item (1.a). When implementing in Maude, `hgct1` and `hgct2` stand for the first and second type of guarded choice.

Now we use rule [1.a1] as an example to make further explanation. From the conditions, we know that the head normal form of P at the location status *null* is `hgct1` and `hgct1` has a component `b&(x := e)(Pi,tag)`. In this case, process P can perform a transition reflecting the execution of that component (i.e., the assignment guarded component). The notation "`env<-(x,e)`" stands for a new state which is the same as `env` except assigning value `e` to `x`.

There are two types of event transitions. When designing the operational semantics, we take the understanding that, if a process has an event transition of the first type, it can also have an event transition of the second type, and vice-versa. When mechanizing the derivation of operational semantics, we take the understanding of regarding a system as closed. Therefore, the second type of event transition is not listed here.

Item (1.b) models the case that the head normal form of a program is expressed as the second type of guarded choice (i.e., event guarded choice). For this case, the program can perform an event transition, including the event transition that one of the guards is fired (i.e., [1.b2]), or the event transition that none of the guards are satisfied (i.e., [1.b1]). The program can also have time delay transition (i.e., [1.b3]), expressed as "`<P,env,#> => <P,env,#>`" in Maude. The notation "`fire(g)(env,env')`" in [1.b2] means that the change from state `env` to `evnv'` can fire the event guard `g`. The notation "`not fire(guard(hgct2))(env,env')`" in [1.b1] means that the state change from state `env` to `evnv'` cannot fire any guards in `hgct2` (i.e., the event guarded choice part of P).

For item (1.c), it models the case that a program is expressed as the time delay guarded choice. For this case, the program can perform time delay transition. It can also have event transition.

For item (1.d), it models the case that the head normal form of a program is expressed as the fourth type of guarded choice, i.e., the compound of the first and second type of guarded choice. For this case, the program can perform instantaneous transition (i.e., [1.d1], [1.d1'] and [1.d2]). It can also have event transition (i.e., [1.d3] and [1.d4]). As the behavior of assignment is instantaneous, the program cannot perform time delay transition. Transition [1.d1] (and [1.d1'])

models the case that the process executes an assignment, whereas transition [1.d2] models the case that the process executes as assignment guard. [1.d3] models the event transition that all the event guards cannot be fired, whereas [1.d4] models the event transition that one guard is fired among all event guards.

Item (1.e) models the case that the head normal form of a program is expressed as the compound of the second and third type of guarded choice. At this case, the program can perform event transition based on firing condition of all guards. The program can also have time delay transition.

Items (2.a) and (2.b) model the situation that a process has already performed a sequence of instantaneous actions for an atomic action. Item (2.a) models the case that the process continues to execute the next instantaneous action for the atomic action. Therefore, it can perform the second type of instantaneous action, leaving the location status before and after the transition unchanged.

Now we consider a single threaded process (denoted by location status $\langle\rangle$), which performs a sequence of instantaneous actions and reaches at the point that the remaining process is guard, time delay or parallel process. In this case, for the head normal form of the remaining process, the subsequent process after each head cannot be $\langle\rangle$. For this case, the original process will perform the third type of instantaneous transitions, i.e., completing an atomic action. This case is illustrated in item (3.a).

Example 5.2 We take program P in Example 4.1 to illustrate the effectiveness of our derivation strategy. Assume that the initial values of x, y are both 0. The head normal form of program P was discussed in Example 4.1. We use the command "`search`" to get its transitions in Maude as following. But the display of the result of "`search`" is in a breadth-first style which is not very straightforward to see. We then use the "`show path` (state number)" command to show one of the path below. In order to display the result neatly, we also omit the rules used by the corresponding transition.

$$
\begin{array}{l}
< \ ((x := x + 1 \ ; \ @(\uparrow y)) \ \| \ y := y + 1) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ empty \ , \# \ > \\
< \ (x := x + 1 \ ; \ @(\uparrow y)) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ empty \ , \ (y, 1) \ , 0 \ > \\
< \ (x := x + 1 \ ; \ @(\uparrow y)) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ (y, 1) \ , \ \# \ > \\
< \ @(\uparrow y) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ (y, 1) \ , \ (y, 1)|(x, 1) \ , \ \langle 1 \rangle \ > \\
< \ @(\uparrow y) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ (y, 1) \ , \ (y, 1)|(x, 1) \ , \ 0 \ > \\
< \ @(\uparrow y) \ \| \ (\#1 \ ; \ y := y + 1) \ , \ (x, 1)|(y, 1) \ , \ \# \ > \\
< \ @(\uparrow y) \ \| \ \ y := y + 1 \ , \ (x, 1)|(y, 1) \ , \ \# \ > \\
< \ @(\uparrow y) \ , \ (x, 1)|(y, 1) \ , \ (x, 1)|(y, 2) \ , \ 0 \ > \\
< \ nil \ , \ (x, 1)|(y, 2) \ , \ \# \ >
\end{array}
$$

The above is one execution sequence leading program P to the terminating state and the final state of variables is "$x = 1 \wedge y = 2$". For program P, there are two execution sequences leading program P to the terminating state. For another execution sequence, the final variable state is also "$x = 1 \wedge y = 2$". □

5.3 Mechanizing Operational Semantics

In the last subsection we provided the strategy for deriving the operational semantics from algebraic semantics. Our approach is via the concept of head normal form. Based on the derivation strategy, we can derive the full set of operational semantics for Verilog as theorems by strict proof. Now we consider the

practical aspect of the derived operational semantics. We apply Maude in mechanizing the derived operational semantics. We select assignment, event guard and parallel composition here for illustrating the mechanization.

As the execution of $x := e$ is instantaneous, if $x := e$ is the first statement of an atomic action, it can be scheduled at once or the environment is allowed to perform some atomic actions. If $x := e$ is not the first action of an atomic action, it should be scheduled to execute at once without interruption by the environment. Time cannot advance for assignment. When animating the operational semantics, we do not know the environment's behaviour. Therefore, we take the understanding of regarding a system as closed. Although the second type of event transition can be derived, we regard it as not executable (see below for $x := e$ and $@(g)$). We use the keyword "`[nonexec]`" to show this.

```
rl : . < x := e , env , # > => < nil , env , env <- (x , e) , 1 , <> > .
rl : . < x := e , env , env' , 1 , <> > => < nil , env , env <- (x , e) , 1 , <> > .
rl : . < x := e , env , env' , 0 > => < x := e , env' , # > .
rl : . < x := e , env , # > => < x := e , env' , # > [nonexec] .
```

The guard $@(g)$ can be immediately fired after it is scheduled to execute; it is actually triggered by the execution of its previous action that has just completed. Another case is that the guard waits to be fired by its environment. Time can also advance before the guard becomes enabled.

```
rl : . < @(g) , env , env' , 1 , <> > => < @(g) , env , env' , 0 > .
rl : . < @(g) , env , env' , 0 > => < nil , env' , # > if fire(g)(env , env') .
rl : . < @(g) , env , env' , 0 > => < @(g) , env' , # > if not fire(g)(env , env') .
rl : . < @(g) , env , # > => < nil , env' , # > if fire(g)(env , env') [nonexec] .
rl : . < @(g) , env , # > => < @(g) , env' , # > if not fire(g)(env , env') [nonexec] .
rl : . < @(g) , env , # > => < @(g) , env , # > .
```

Now we consider the mechanizing of the derived operational semantics for parallel composition. If one of the two parallel parts of a Verilog program can perform the first instantaneous action of an atomic action, then the whole process can also make this transition.[4]

```
crl : . < P || Q , env , # > => < par(nil,Q) , env , env' , 0 >
    if . < P , env , # > => < nil , env , env' , 1 , seq > .
crl : . < Q || P , env , # > => < par(Q,nil) , env , env' , 0 >
    if . < P , env , # > => < nil , env , env' , 1 , seq > .
crl : . < P || Q , env , # > => < par(P',Q) , env , env' , 1 , <1> ^ seq >
    if . < P , env , # > => < P' , env , env' , 1 , seq > /\ P' =/= nil .
crl : . < Q || P , env , # > => < par(Q,P') , env , env' , 1 , <2> ^ seq >
    if . < P , env , # > => < P' , env , env' , 1 , seq > /\ P' =/= nil .
```

If one of the two parallel parts of a Verilog program continues to perform the instantaneous action of an atomic action, then the whole process can also make this transition.

```
crl : . < P || Q , env , env' , 1 , <1> ^ seq > => < par(P',Q) , env , env'',<1> ^ seq >
    if . < P , env , env' , 1 , seq > => < P' , env , env'' , 1 , seq > /\ P' =/= nil .
crl : . < Q || P , env , env' , 1 , <2> ^ seq > => < par(Q,P') , env , env'',<2> ^ seq >
    if . < P , env , env' , 1 , seq > => < P' , env , env'' , 1 , seq > /\ P' =/= nil .
```

[4] For this case, we can also have the situation that P or Q may be the empty process. In the consideration for other cases below, for $P \parallel Q$, P or Q maybe also be the empty process. We omit the transition rules which are similar to the normal situation.

If one of the two parallel parts of a Verilog program exits from an atomic action, then the whole process can also exit from the atomic action. A parallel process can also exit from its prior instantaneous section.

```
crl : . < P || Q , env , env' , 1 , <1> ^ seq > => < par(P',Q) , env , env' , 0 >
    if . < P , env , env' , 1 , seq > => < P' , env , env' , 0 > .
crl : . < Q || P , env , env' , 1 , <2> ^ seq > => < par(Q,P') , env , env' , 0 >
    if . < P , env , env' , 1 , seq > => < P' , env , env' , 0 > .
crl : . < P || Q , env , env' , 1 , <> > => < P || Q , env , env' , 0 > .
```

If one of the two parallel parts of a Verilog program executes an atomic assignment, then the whole process can also execute the atomic assignment.

```
crl : . < P || Q , env , # > => < par(P',Q) , env , env' , 0 >
    if . < P , env , # > => < P' , env , env' , 0 > .
crl : . < Q || P , env , # > => < par(Q,P') , env , env' , 0 >
    if . < P , env , # > => < P' , env , env' , 0 > .
```

$P \parallel Q$ can perform a triggered action caused by its predecessor or one of its components. $P \parallel Q$ allows the environment to perform an atomic action. $P \parallel Q$ allows time advance **iff** both components do so.

```
crl : . < P || Q , env , env' , 0 > => < par(P',Q') , env' , # >
    if . < P , env , env' , 0 > => < P' , env' , # > /\
       . < Q , env , env' , 0 > => < Q' , env' , # > .
crl : . < P || Q , env , # > => < par(P',Q') , env' , # >
    if . < P , env , # > => < P' , env' , # > /\ . < Q , env , # > => < Q' , env' , # > .
crl : . < P || Q , env , # > => < par(P',Q') , env , # >
    if . < P , env , # > => < P' , env , # > /\ . < Q , env , # > => < Q' , env , # > .
```

Example 5.3 Let P be the program described in Example 4.1 and Example 5.2. In Example 5.2, we have already considered the execution sequence of program P using the derivation strategy via algebraic semantics. Now we consider its execution based on the transition rules (i.e., the operational semantics in this section).

There are also two execution sequences leading program P to the terminating state. The first sequence is the same as the one described as Example 5.2 and the final state of program variables is also "$x = 1 \wedge y = 2$".

For the second execution sequence, the final state of program variables is also "$x = 1 \wedge y = 2$" and its detailed transition is as below. This execution sequence is the same as the second sequence in Example 5.2 leading program P to the terminating state (although we didn't list it).

$$
\begin{array}{l}
< \ ((x := x + 1 \ ; \ @(\uparrow y)) \ \parallel \ y := y + 1) \ \parallel \ (\#1 \ ; \ y := y + 1) \ , \ empty \ , \# \ > \\
< \ ((@(\uparrow y) \ \parallel \ y := y + 1) \ \parallel \ (\#1 \ ; \ y := y + 1) \ , \ empty \ , \ (x, 1) \ , \ \langle 1 \rangle \ > \\
< \ ((@(\uparrow y) \ \parallel \ y := y + 1) \ \parallel \ (\#1 \ ; \ y := y + 1) \ , \ empty \ , \ (x, 1) \ , \ 0 \ > \\
< \ (@(\uparrow y) \ \parallel \ y := y + 1) \ \parallel \ (\#1 \ ; \ y := y + 1) \ , \ (x, 1) \ , \ \# \ > \\
< \ @(\uparrow y) \ \parallel \ (\#1 \ ; \ y := y + 1) \ , \ (x, 1) \ , \ (x, 1)|(y, 1), \ 0 \ > \\
< \ \#1 \ ; \ y := y + 1 \ , \ (x, 1)|(y, 1), \ \# \ > \\
< \ y := y + 1 \ , \ (x, 1)|(y, 1), \ \# \ > \\
< \ nil \ , \ (x, 1)|(y, 2), \ 1 \ , \ \langle\rangle \ >
\end{array}
$$

The mechanical approach indicates that the transition system from the derived operational semantics is the same as the one from the derivation strategy. □

6 Conclusion and Future Work

This paper has presented how an algebraic semantics links with the operational semantics for Verilog, starting from the algebraic semantics. Our approach is to derive the operational semantics from the algebraic semantics. The mechanical method is applied in linking the two semantics. We used the equational and rewriting logic system Maude to support the mechanical implementation.

We have given the algebraic laws. Our approach is new, where a process is expressed as the guarded choice of a set of guarded components with location status. This guarded choice gives us a way to express how a process can be sequentialized, which also reflects the scheduling policy. In order to support the derivation, we introduced the concept of head normal form for every program at a location status. We have studied the derivation of the operational semantics for Verilog from its algebraic semantics. We have given the definition of the derivation strategy. Then a transition system (i.e., operational semantics) for Verilog can be derived via the derivation strategy. The algebraic laws, head normal form, derivation strategy and derived transition system are all implemented in the Maude system. The results mechanized indicate that the transition system of the derived operational semantics is the same as the one based on the derivation strategy.

Semantic linking is the challenging research [7]. For the future, we are continuing to explore further linking theories for Verilog semantics [8]. In particular the mechanical approach to the derivation of denotational semantics from algebraic semantics is also very challenging.

Acknowledgement This work is supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61061130541 and No. 61021004), and Shanghai Leading Academic Discipline Project (No. B412).

References

1. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proc. RTA 2003: 14th International Conference on Rewriting Techniques and Applications*, Valencia, Spain, June 9-11, 2003, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, June 2003.
2. M. Clavel, F. F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.6)*. January 2011.
3. M. J. C. Gordon. The semantic challenge of Verilog HDL. In *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 136–145. IEEE Computer Society Press, June 1995.
4. M. J. C. Gordon. Relating event and trace semantics of hardware description languages. *The Computer Journal*, 45(1):27–36, 2002.
5. J. He and Q. Xu. An operational semantics of a simulator algorithm. Technical Report 204, UNU/IIST, P.O. Box 3058, Macau SAR, China, 2000.

6. J. He and H. Zhu. Formalising Verilog. In *Proc. ICECS 2000: IEEE International Conference on Electronics, Circuits and Systems*, pages 412–415. IEEE Computer Society Press, December 2000.
7. C. A. R. Hoare. Algebra of concurrent programming. In *Meeting 52 of WG 2.3*, 2011.
8. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.
9. IEEE. *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, volume IEEE Standard 1364-1995. IEEE, 1995.
10. IEEE. *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language*, volume IEEE Standard 1364-2001. IEEE, 2001.
11. Y. Li and J. He. Formalising Verilog: Operational semantics and bisimulation. Technical Report 217, UNU/IIST, P.O. Box 3058, Macau SAR, China, November 2000.
12. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science*, 4:190–225, 1996.
13. N. Martí-Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
14. J. Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*. To appear.
15. R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1990.
16. R. Milner. *Communication and Mobile System: π-calculus*. Cambridge University Press, 1999.
17. N. Nissanke. *Realtime Systems*. Prentice Hall International Series in Computer Science, 1997.
18. A. Verdejo and N. Martí-Oliet. Implementing ccs in maude 2. *Electronic Notes in Theoretical Computer Science*, 71:282–300, 2002.
19. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
20. H. Zhu, J. P. Bowen, and J. He. Deriving operational semantics from denotational semantics for Verilog. In *Proc. APSEC 2001: 8th Asia-Pacific Software Engineering Conference*, pages 177–184. IEEE Computer Society Press, December 2001.
21. H. Zhu, J. P. Bowen, and J. He. From operational semantics to denotational semantics for Verilog. In *Proc. CHARME 2001: 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 449–464. Springer-Verlag, September 2001.
22. H. Zhu and J. He. A semantics of Verilog using Duration Calculus. In *Proc. International Conference on Software: Theory and Practice*, pages 421–432, August 2000.
23. H. Zhu, J. He, and J. P. Bowen. From algebraic semantics to denotational semantics for verilog. *Innovations in Systems and Software Engineering: A NASA Journal*, 4(4):341–360, 2008.