MEMOCODE'10, IEEE, pp 179-188. 2010

A Formal Executable Semantics of Verilog*

Patrick Meredith Michael Katelman José Meseguer Grigore Roșu
Department of Computer Science
University of Illinois at Urbana-Champaign
{pmeredit, katelman, meseguer, grosu}@uiuc.edu

Abstract

This paper describes a formal executable semantics *for the Verilog hardware description language. The goal of our formalization is to provide a concise and mathematically rigorous reference augmenting the prose of the official language standard, and ultimately to aid developers of Verilog-based tools; e.g., simulators, test generators, and verification tools. Our semantics applies equally well to both* synthesizeable *and* behavioral *designs and is given in a familiar, operational-style within a logic providing important additional benefits above and beyond static formalization. In particular, it is* executable *and* searchable *so that one can ask questions about how a, possibly nondeterministic, Verilog program can legally behave under the formalization. The formalization should not be seen as the final word on Verilog, but rather as a starting point and basis for community discussions on the Verilog semantics.*

1 Introduction

There are many tools based on Verilog, which implicitly, through their implementation, interpret the semantics of the language; *e.g.,* simulators, test generators, and formal verification tools. The language standard for Verilog [7] is the only official document regarding the meaning of Verilog, and anyone implementing a Verilog-based tool should understand it thoroughly. It is written in English prose and, while extensive and generally clear, we have encountered important situations where the intentions of the standard were unclear and we had no good means of resolving our conundrum.

One of the goals of a *formal semantics* is to avoid problems with the imprecise nature of prose, by using rigorous mathematical definitions. Therefore, in this paper we develop an extensive formalization of the Verilog language, using a familiar, operational-style. Our goal is not to replace the standard, but rather to *augment* it with a formal, yet intuitive and operationally clean, description of the language that tool designers and other interested parties can use to help resolve complex questions about the language, when they arise. This is useful for all types of tools, but *especially* for formal verification tools, where the advertised guarantees are very strong.

There are many common methods of giving a formal semantics, such as structural-operational semantics [15, 16], reduction semantics with evaluation contexts [4], and denotational semantics [19]. In this paper we use an operational-style approach called *rewriting logic semantics (RLS)* [12, 3], and within RLS a definitional technique called *K* [17]. While it is outside the scope of this paper to give a detailed comparison of our approach with the many others (for the interested reader, see [12, 3, 17]), there are three benefits of the chosen approach that we briefly mention. First, rewriting logic semantics admits a style similar to functional programming, which is familiar to many people. Second, the semantics is concurrent, and can directly support descriptions of nondeterministic computations. Third, there are tools, such as Maude [2], allowing us to *execute* Verilog programs *directly* within the rewriting logic semantics. No separate interpreter need be written, and we can even systematically *search* through the different concurrent interleavings allowed by the inherently parallel semantics of Verilog.

In principle, there is no a priori reason why the semantics given in this paper could not be used as the basis for a formal semantics within another framework, should one desire that, provided that the framework in question also supports concurrency. Therefore, the goals of this paper do not depend on a definitional style; it is simply necessary to make *some* choice, and we believe that there are real benefits, such as the ones described above, and used to good effect in subsequent sections, to a rewriting logic semantics. Those interested in translating a rewriting logic semantics to other styles may find good guidance in [3].

In addition to the description of our formal semantics given in this paper, and its full definition in rewrit-

*Supported in part by NSF grants CCF-0916893, CNS-0720512, CCF-0905584, and CCF-0448501, by NASA contract NNL08AA23C, by a Samsung SAIT grant, and by several Microsoft gifts.

MEMOCODE'10, IEEE, pp 179-188. 2010

```
1  module flipflop(clk, in, out);
2    input clk
3    input[15:0] in;
4    output[15:0] out;
5
6    reg[15:0] r;
7
8    assign out = r;
9
10   always@(posedge clk)
11   begin
12     r <= in;
13   end
14 endmodule
```

Figure 1. Flpflop Example

ing logic [8, 10], we contribute observations about particularly curious aspects of Verilog, some where we have made choices of interpretation of the standard where other interpretations are possible. We also show how our definition can be used with the Maude tool to ask questions about the meaning of Verilog programs.

We hope that our semantics and observations can provide a good basis for a discussion within the community of Verilog users and tool developers, one ultimately resulting in more robust designs, consistency between tools, and simpler construction of new tools based on Verilog. In fact, questions as to the correct semantics of Verilog can be seen in actual production systems. During the development of our semantics we ran into situations in which two popular simulators, VCS [20] and Icarus Verilog (Iverilog) [23], differ in the evaluation of fairly small Verilog programs. The value of a formal semantics for these situations is to provide a precise interpretation of the Verilog standard that, although open to challenges itself, can clarify the discrepancies between different tools where source code may sometimes be proprietary. Section 5 discusses one of these examples, though we have found several others [9].

The rest of this paper is organized as follows: Section 2 gives a high-level overview of the Verilog semantics and the main concepts it involves, and also discusses some curious aspects we have encountered, including bugs in well-known Verilog simulators. Section 3 gives a brief overview of rewriting semantics, followed in Section 4 with a guided-tour of our semantic definition. The full definition "source code" can be found at [9]. Section 6 compares our definition to some other definitions of Verilog. Section 5 discusses how our semantics should be used to help resolve questions about Verilog, as well as some additional possible uses of the semantics. Finally, Section 7 concludes.

2 Verilog Semantics: High-Level Concepts

While we believe most of our readers will be familiar with Verilog, we begin with a simple example to show some of the features of the language. Then we introduce some of the high level aspects of the semantics of Verilog: the different types of assignments, processes and events, process and event scheduling, and value sizing.

2.1 An Introductory Example

Figure 1 shows a Verilog *module* that defines a simple flipflop. A module is a unit of design that allows for code reuse and abstraction. It is similar in spirit to modules from several programming languages. While the example is simple, it illustrates some important features of Verilog.

Verilog has two main types of data: *variables* and *nets*. Variables abstract the idea of state, requiring storage[1], while nets that of wires that carry information from one area of a design to another. The **`input`** and **`output`** keywords declare which variables are inputs and outputs to the module. Module inputs and outputs are automatically assumed to have the net type **`wire`**, so it is not necessary to give them a type declaration unless a different type is wanted. The name **`r`** is declared with the **`reg`** type, which is a variable type.

Both nets and variables are either single bits (by default) or bit vectors of more than one bit. The input **`clk`** is a single bit value, because no indices are specified for it, while **`in`** is declared to be sixteen bits indexed from **`15`** to **`0`**. According to the Verilog standard, variable data types may only be assigned within procedural blocks, such as the one on lines 10–13, which will be described below, while net types can only be assigned in wired assignments such as the one on line 8, which will also be described below.

Lines 10-13 show a procedural block, in this case an **`always`** block. An **`always`** block denotes a constantly running computation, essentially an infinite loop. Note that Verilog also has an **`initial`** block, which runs only at the beginning of simulation. An **`initial`** block can be seen in the example in Figure 2. The term procedural blocks refers to **`always`** and **`initial`** blocks collectively. In its most basic form, the **`always`** block takes a single statement. The phrase **`@(posedge clk)`** delays the statement or block following it until its condition is met. Meeting the condition, in this case, requires the value of **`clk`** to change from some non-**`1`** value to **`1`** (a positive edge). This allows us to delay the entire body of the **`always`**, resulting in the value of the **`reg r`** only changing on the positive edge of the **`clk`** input. For the purposes of this definition, we refer to **`@(`**X**`)`** for some X as a *trigger*, while X itself is often known as a *sensitivity list* (X may refer to more than one variable or net separated by the keyword **`or`**).

The assignment on line 8 is a **`net`** assignment. Perhaps somewhat counter-intuitively, this assignment will be the last action of the module on a given positive edge of **`clk`**.

[1] It must be noted that, when synthesized to hardware, data declared with a variable type may require no actual state elements. If so, the synthesizer will allocate no storage.

```
1  module assignment_types(clk, in);
2    input clk;
3    input [15:0] in;
4    reg[15:0] b_1, b_2, nb_1, nb_2;
5
6    initial nb_1 = 0;
7
8    always@(posedge clk)
9    begin
10      b_1 = in;
11      b_2 = b_1 + 1;
12   end
13
14   always@(posedge clk)
15   begin
16     nb_1 <= in;
17     nb_2 <= nb_1 + 1;
18   end
19 endmodule
```

Figure 2. Assignment Types Example

A **net** assignment occurs whenever any value in its right hand side changes, in this case, when **r** changes. This can be thought of in terms of hardware as attaching a wire to the output of the **reg r**.

This example only illustrates a very small number of Verilog features, to prepare the reader for the rest of the paper. More features will be introduced as needed.

2.2 Types of Assignments

As the example in Figure 1 illustrates, there are two basic types of assignments at the top level, *continuous* assignments, such as the one on line 8, that allow assignment to net types, and procedural assignments, such as the one on line 12, that allow assignment to variable types. Procedural assignments can be broken down further into *blocking* and *non-blocking* assignments.

The module in Figure 2 shows an **initial** block and two **always** blocks, which look very similar, yet compute very different results. In the **initial** block, **nb_1** is initialized to **0**. In the block on lines 8–12, blocking assignments are used (hence the variable names **b_1**,**b_2**), while lines 14–18 use non-blocking assignments.

To understand what is going on in this example, let us assume a value for **in**, say **1**. Then, considering the first **always** block on lines 8–12, the value of **b_1** will be **1**, while that of **b_2** will be **2**. This is because the assignment of **b_1** *blocks* the statements following it until its completion. The non-blocking assignments in the block on line 14–18 do not block the statements following them: **nb_1** will contain **1**, but **nb_2** will also contain **1**, because the *previous* value of **nb_1** (**0**) is used in the assignment on line 17.

2.3 Processes and Events

As a language created to model and design circuits, Verilog is inherently concurrent. Capturing this concurrency, and the resulting non-determinism allowed by the standard, is one of the most important tasks of any formal definition of the language. Many Verilog users, however, learn the language primarily through simulators, which tend to be single threaded and deterministic.

To ease understanding and maintain consistency, we adopt several of the terms used in the Verilog standard. First and foremost is that of a *process*. In a Verilog design, a process is anything that can perform computation. According to Section 11.2 in the standard, "Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs". Going back to our introductory **flipflop** example from Figure 1, the **always** block on lines 10–13 is one process, while the wire assignment on line 8 is another. The **module** itself is also a process.[2] Section 4 discusses our process formalization.

While processes are very specific, the terminology of *event* encompasses several different concepts. Except where specifically mentioned, we try to make the event terminology of the standard explicit in the definition, to ease understanding for those familiar with the standard. Every update of a net or variable is an *update event*. The evaluation of a process is an *evaluation event*. This is the only type of event that is not explicitly represented in the definition, which merges the concepts of process and evaluation event, effectively treating processes as events.

2.4 Event and Process Scheduling and Timing

While Verilog is used to synthesize circuit designs, it is essentially designed for simulation. Because of this, Verilog is sensitive to simulator time. In fact, in addition to the ability to delay statements until a particular condition holds, such as on line 10 in Figure 1, it is also possible to delay statements some number of simulator cycles.[3] The syntax for this consists of preceding a statement, say S, with **# N**, which means that S will be delayed **N** simulator cycles.

Events can be further divided into five categories that determine how they are scheduled for execution with respect to simulation time: *active events*, *inactive events*, *non-blocking assign update events*, *monitor events*, and *future events*. We further add the category of *listening events*, which do not exist in the standard but help clarify the execution of Verilog designs.

Active events are all events that are currently running, *i.e.,* they are not waiting for any specific trigger, and they have not been delayed. Inactive events are curious, in that they only occur when a statement has been delayed by exactly zero simulator cycles (*e.g.,* **# 0 r = 1;**). Non-

[2] Although our definition handles modules much as typical simulators (it inlines them into the top module), this semantics gives equivalent results to treating modules as actual processes.

[3] Simulator cycles are an abstract unit of time not tied to any particular real world unit of time.

```
module value_size;
  reg[3:0] r_1, r_2;
  reg[15:0] out;

  initial
  begin
    r_1 = 15;
    r_2 = 15;
    out = r_1 + r_2;
  end
endmodule
```

Figure 3. Value Sizing Example

blocking assign update events correspond to the actual change in state that occurs after a non-blocking assignment. Monitor events are related the Verilog *monitor* statement, which is essentially a print statement that occurs at the end of every simulator cycle in which its arguments *change*[4]. Future events are processes that have been delayed by some non-zero amount, which must still eventually execute. Listening events are those events that are waiting for a particular trigger to occur; they will be promoted to active events/processes as soon as that trigger occurs (such as the positive edge of `clk` on line 8 of Figure 2).

Each type of event, as listed, is promoted to an active event/process only when there are no events before it in the list, except for listening events, which may be promoted as soon as the trigger that they are listening for occurs. For example, inactive events are all, *at the same time*, activated (that is, promoted to active events) when there are no more active events/processes left. Similarly, non-blocking assign update events are all simultaneously promoted to active events/processes when there are no more active or inactive events in the given simulation cycle. When all events, except for listening events and future events, have been exhausted, the time is advanced to the time specified for the earliest future event. If there are no pending future events the program completes execution. Section 4.6 provides the formal definition of part of this scheduling.

2.5 Value Sizing

Verilog has interesting rules for the size of operands. Figure 3 shows a simple, by no means exhaustive, example of this. Despite the fact that both `r_1` and `r_2` are only four bits wide, the `reg out` is still assigned the value `30`. For the purposes of the addition, `r_1` and `r_2` are treated as sixteen bit quantities, because `out` is a sixteen bit quantity. There are many different rules for the sizing of values; the above example only covers one of them (sizing to the left hand side of an expression). All of the rules are covered in our definition. Due to the fact that the standard specifies these rules very clearly, we refer the interested reader to the standard or the full specification of our definition at [9].

[4] It ignores changes in the output of the `$time()` function, however.

3 Rewriting Logic Semantics

To better understand the semantics of Verilog presented in this paper, we provide a brief introduction to term rewriting, rewriting logic, and the use of rewriting logic in programming language definitions. Term rewriting is a standard computational model supported by many systems; rewriting logic [10, 8] organizes term rewriting modulo equations as a logic and serves as a foundation for programming language semantics [11, 12, 3]. K [17], a form of rewriting logic semantics based on continuations that is adopted in this paper, provides explicit representations of the program/design control context and represents the state of a running program/design as a multiset of nested terms.

3.1 Term Rewriting

Term rewriting is a model of computation that works by progressively changing (rewriting) a term. This rewriting process is defined by a number of rules of the form $l \to r$, where the terms l, r may contain variables. One step of rewriting is performed by first finding a rule whose left hand side matches either the entire term or a subterm. This is done by finding a substitution, θ, from variables to terms such that the left-hand side of the rule, l, matches part or all of the current term when the variables in l are instantiated according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand side r. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution as above. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, may not terminate.

There exist a plethora of term rewriting engines, including ASF [21], Elan [1], Maude [2], OBJ [5], Stratego [22], Tom [13], and others. Rewriting is also a fundamental part of many existing languages and theorem provers.

3.2 Rewriting Logic

Rewriting logic [10] is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the λ calculus, where λ terms can be grouped into equivalence classes based on relations such as α and β equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency, where different orders of evaluation could lead to non-equivalent results, such as in the case of data races. Transitions in the system, such as updating a variable or net in Verilog, must be with rules, rather than equations, because they result in actual changes of evaluation state. Simple evaluation, such as adding two numbers, can be written using an equation because no change to the externally visible state occurs.

The distinction between rules and equations is crucial for formal analysis, since terms which are equal according to equational deduction can all be collapsed into the same analysis state. Rewriting logic is connected to term rewriting in that some equations, such as associativity, commutativity, and identity, can be used as *structural axioms*, so that matching of rules happens *modulo* such axioms, and the remaining equations and rules, respectively of the form $l = r$ and $l \Rightarrow r$, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming both into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term and "executing" it.

3.3 Rewriting Logic Semantics and K

This paper is part of the *rewriting logic semantics (RLS)* project [12], whose aim is to formalize the semantics of programming or description languages within rewrite logic, with the explicit purpose of making such semantics *useful*. For example, one can use an RLS to execute programs directly within their mathematical definition, or to formally analyze programs using general-purpose RLS tools, or even to prove meta-theorems about the defined languages. Several such languages have already been given an RLS; we cannot list them here due to space constraints, but we refer the interested reader to the RLS project (see [11, 12, 3] and the references there). RLS does not enforce any particular definitional style or technique; in particular, as shown in [3], various existing techniques can be framed as particular RLS approaches. The K technique [17] is adopted in this paper, because it makes it particularly convenient to deal with complex configurations and concurrent language constructs, as needed for the semantics of Verilog.

In K, the current configuration/state is represented as a multiset of nested terms representing the current processes, environment(s), pending update events, etc. Information stored in the state can be nested, allowing logically related information to be grouped and manipulated as a whole. One of the most important pieces of information are the *continuations*, named k, which are first-order representations of the current computation for each active process, made up of a list of instructions separated by $\curvearrowright$. The continuation can be seen as a stack, with the current instruction at the left and the remainder (continuation) of the computation to the right. These stacks, along with other state components, can be saved and restored later, allowing in our case for a clear definition of the delay and trigger constructs. Below is a structural rule[5] specifying how blocks denoted by **begin** and **end** are scheduled for evaluation within a process:

$$\begin{aligned}&\mathsf{k}(\mathsf{stmt}(\mathbf{begin}\ S\ SL\ \mathbf{end} \curvearrowright K)\\ &= \mathsf{k}(\mathsf{stmt}(S) \curvearrowright \mathsf{stmt}(\mathbf{begin}\ SL\ \mathbf{end}) \curvearrowright K)\end{aligned}$$

The conventions used in the above sentence will be maintained throughout this paper: semantic operators (such as stmt, k) are denoted in a sans serif font, while portions of Verilog syntax (**begin** and **end**) are denoted in bold face, and term variables (such as S and SL) are in italics. This equation unrolls a single statement consisting of a block into a semantic series of statements. The variable S refers to a single statement, while SL is a list of statements and K refers to the rest of the computation. In our syntax it is the statement itself that ends with Verilog's semi-colon terminator, so statement lists use a space for concatenation of statements. This is an equation (or a structural rule in K) rather than a (computational) rule because no change to the actual program state takes place; only the representation of the program changes. Below is a computation for the block on lines 9–12 of Figure 2 before and after the above equation is applied one time, and then after it is applied again:

$$\begin{aligned}&\mathsf{k}(\mathsf{stmt}(\mathbf{begin\ b\_1{=}in;\ b\_2{=}b\_1{+}1;\ end}))\\ &= \mathsf{k}(\mathsf{stmt}(\mathbf{b\_1{=}in;}) \curvearrowright \mathsf{stmt}(\mathbf{begin\ b\_2{=}b\_1{+}1;\ end}))\\ &= \mathsf{k}(\mathsf{stmt}(\mathbf{b\_1{=}in;}) \curvearrowright \mathsf{stmt}(\mathbf{b\_2{=}b\_1{+}1;})\\ &\qquad\qquad \curvearrowright \mathsf{stmt}(\mathbf{begin}\ \mathsf{empty}\ \mathbf{end}))\end{aligned}$$

Note that this is the whole computation, so the K in the above equation matches an empty computation (which is the identity for the $\curvearrowright$ operator). As the sans serif font denotes, empty is a semantic constant. It simply stands for an empty statement list, the equation below handles this case:

$$\mathsf{stmt}(\mathbf{begin}\ \mathsf{empty}\ \mathbf{end}) = \mathsf{empty}$$

For simplicity we abuse the term empty to be identity for any sort (*i.e.,* for both statement lists and computations).

4 Rewriting Logic Semantics of Verilog

We next introduce the most interesting portions of our definition. We have to present chosen those features of Verilog that we feel are most central to the spirit of the language. Other features are elided, but not from the definition itself, because either we feel the standard describes them very clearly (such as value sizing mentioned before), or they are not particularly interesting (such as the equations and rules for the Verilog **$display()** function). Interested readers are encouraged to check the full definition [9].

```
env(clk ← [0#1], in ← [0#16], out ← [0#16], r ← [0#16])
time(0)
activeProcesses(k(top(always@(posedge clk)
                      begin
                        r  <= in;
                      end
                      )))
updateEvents(empty)
inactiveEvents(empty)
nonBlockingAssignUpdateEvents(empty)
monitorEvents(empty)
futureEvents(empty)
futureMonitorEvents(empty)
listeningEvents(continuousListeningEvent(r, out, exp(16, r)))
output(empty)
finish(false)
```

Figure 4. Initial Configuration for Flipflop

4.1 Verilog Configuration

A *configuration* is the term representation for the state of a whole program. All equations and rules are repeatedly applied to configurations to advance the computation carried away by the system. The initial configuration for the flipflop example program from Figure 1 can be seen in Figure 4.

The first subterm, env is the environment, *i.e.,* a mapping from variable or net names to bit vectors. The first element in a bit vector is the value, the second (after the #) is the size of the bitvector. Nothing in the configuration distinguishes nets from variables. This is one of the first lessons of the Verilog standard: net and variable types only need be distinguished at the syntactic level. It is enough to have a static analysis that determines if nets and variables are used incorrectly. This fits neatly with the fact that the storage elements for variable types are generally not allocated by Verilog synthesis tools when no storage is actually needed. The distinction between nets and variables is an archaic part of the standard, no longer strictly necessary. It is known that combinatorial circuits can be generated from **always** blocks under certain conditions[6]. Section 4.3 discusses how modifications to the environment are defined.

The second subterm, time, contains the current simulator cycle, which always begins at 0.

The subterm activeProcesses contains all the currently active processes, in this case only one, the **always** block at lines 10–13. As mentioned in Section 3.3, k denotes a computation. The operator top denotes that the enclosed syntax is a *top* term (a name we created). We break intramodule Verilog syntax into three categories: top, statement, and expression. Statement and expression are standard terms (expressions compute values, statements are evaluated for side effects), while top is either a procedural block (**always** or **initial**) or a continuous wire assignment.

Section 4.2 discusses how the evaluation of procedural blocks (**always** and **initial**) is defined, while Section 4.5 will show how the trigger (**@(posedge clk)**) is evaluated.

Subterm inactiveEvents contains empty because no statements have been delayed by zero cycles; this will also be discussed in Section 4.5. Additionally, nonBlockingAssignUpdateEvents, monitorEvents, futureEvents are all empty because no non-blocking assignments, monitor statements, or non-zero delay statements have occurred in the initial state of the system. The subterm futureMonitor events is also related to monitor events. Monitor statements are not covered in depth here (see [9]).

The subterm listeningEvents contains all those processes that are listening for a change in some variable or net, so that they can continue execution. In the initial state, only the **assign** statement is listening. The term continuousListeningEvent distinguishes this as a continuous net assignment, rather than a procedural process, which would use instead the term listeningEvent. The reason for this distinction is explained in Section 4.3. The continuousListeningEvent operator takes three arguments: the first, in this case **r**, is the sensitivity list that must experience a change for the listening event to be activated (in this case it is only a single variable). The second argument is the net to which the continuous net assignment assigns; this is significant for reasons explained in Section 4.3. The last argument is the expression that makes up the right hand side of the net assignment, stored as a computation. The term constructor exp simply denotes that this computation is an expression rather than top or statement. The first argument is used for sizing values, which, as mentioned in Section 2.5, is elided from this paper; the argument is included here for completeness.

The subterm output is used to hold the output of the functions **$display**, **$strobe**, and **$monitor**. The contents of output are reported at program termination; it should obviously begin empty as no output has been generated before the program runs. The operator finish supports the Verilog function **$finish**.

4.2 Procedural Blocks

The semantics for **initial** blocks are very simple. The statements of an **initial** block must run exactly once. The equation below simply strips off the **initial** keyword, forcing the statements represented by S to evaluate:

$$\mathsf{top}(\texttt{initial}\ S) = \mathsf{stmt}(S)$$

The semantics of **always** states that the statements of the body of the block must be repeated indefinitely; thus, the statements S are run, but another copy of the **always** block is also scheduled to run after S completes:

[5] The K framework [17] allows two kinds of semantic rules, structural and computational. Since we here use the K technique within rewrite logic, we can represent K's structural rules as rewrite logic equations.

[6] The **always** block begins with a trigger statement that contains all the names of every variable or net used to compute values, and no persistent storage is necessary.

$$\mathsf{k}(\mathsf{top}(\texttt{always}\ S)) = \mathsf{k}(\mathsf{stmt}(S) \curvearrowright \mathsf{top}(\texttt{always}\ S))$$

Note that the equation matches the k operator to keep the **always** block from infinitely unrolling rather than executing the statements of the body before unrolling another step.

4.3 Assignments

At their most basic level, the different types of assignments generate update events. The update events themselves are responsible for actually updating the environment and waking up any listening processes, as will be explained in the next section. In addition to the presented equations/rules, there also are assignments with delays or triggers in the right hand side. The semantics of these differ from delayed statements, which are described in Section 4.5, but are elided for the sake of brevity (available at [9]).

We begin with blocking assignments, for which it is imperative that any trailing statements are not executed until the generated update event completes. How to calculate the value of the right hand side of the assignment is easy and not shown. The equation below assumes it evaluated, the result being the *BV* at the beginning of the computation:

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(BV \curvearrowright \mathsf{blockingAssign}(X) \curvearrowright K)\ PS)\\ \mathsf{updateEvents}(ES)\\ = \mathsf{activeProcesses}(PS)\\ \mathsf{updateEvents}(\mathsf{updateEventList}(\mathsf{updateEvent}(X, BV), K)\ ES)\end{array}$$

Here *PS* is a process set, *ES* is an event set, X is a variable name, and K is the rest of the computation. The important point here is that K is placed in the updateEvent list that is added to updateEvents. This K will contain any remaining statements in the given procedural block. As seen in the next section, this K will be run as an active process once the update event updates the state of the system. K is removed from the activeProcesses for the time being, until the generated updateEvent completes. The updateEventList term allows us to group any number of update events that must be executed in order. This is useful both for assignments with concatenations on the left-hand side (see [9]), and for scheduling non-blocking assign update events.

While very similar in form, we use a rule to define non-blocking assignments. This way all of the non-blocking assignments may be interleaved non-deterministically. If an equation were used, non-blocking assignments would only be allowed to interleave in one order. The assignments added to the nonBlockingAssignUpdateEvents set are eventually scheduled to execute in one updateEventList (see Section 4.6). This is to ensure the mandate that non-blocking assignments in one procedural block complete in order:

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(BV \curvearrowright \mathsf{nonBlockingAssign}(X) \curvearrowright K)\ PS)\\ \mathsf{nonBlockingAssignUpdateEvents}(EL)\\ \rightarrow \mathsf{activeProcesses}(\mathsf{k}(K)\ PS)\\ \mathsf{nonBlockingAssignUpdateEvents}(\mathsf{updateEvent}(X, BV); EL)\ ES)\end{array}$$

```
module netassign;
  wire w;
  reg r;

  assign w = r;

  initial
  begin
    r = 0;
    r = 1;
  end
endmodule
```

Figure 5. Net Assignment Example

Here *EL* represents a list (*i.e.,* an ordered sequence) of events. Note that the term K is allowed to continue as $\mathsf{k}(K)$ appears in the activeProccesses on the right hand side of the rule. This is exactly the desired semantics of a non-blocking assignment: the rest of the block is allowed to complete before the update event from the assignment is allowed to make any change to the environment.

Lastly, we have the two equations for continuous (net) assignment. The important issue here is that only one outstanding update event exists for a given net at a time. This is actually an issue which is not explicitly covered in the standard. The best we can glean from the standard is that a net assignment should perform essentially as an **always** block with one blocking assignment in it (save that the type of the left hand side must be a net type). We argue that this does not quite make sense, though we do provide a version of the definition that performs this way. Instead, we propose a semantics that is similar to the one proposed by Gordon [6]. The issue is that, since update events are processed concurrently, if the **always** block approach is taken, very counter-intuitive results are possible. For example, in Figure 5, if the **always** block semantics is used, **w** could be **0** or **1** after evaluation. Because nets are supposed to represent the ideal of a wire attached to an input, we argue that only **1** makes sense as a result for **w**. An argument can be made that using an **always** block semantics makes sense, and that assignments to the same variable should never occur in the *same* simulator cycle, however. We believe that this is an interesting and open topic of debate.

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{continuousk}(X, BV_1)\ PS)\\ \mathsf{updateEvents}(\mathsf{continuousUpdateEvent}(X, BV_2)\ ES)\\ = \mathsf{activeProcesses}(PS)\\ \mathsf{updateEvents}(\mathsf{continuousUpdateEvent}(X, BV_1)\ ES)\end{array}$$

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{continuousk}(X, BV)\ PS)\\ \mathsf{updateEvents}(ES)\\ = \mathsf{activeProcesses}(PS)\\ \mathsf{updateEvents}(\mathsf{continuousUpdateEvent}(X, BV)\ ES)\\ \textit{otherwise}\end{array}$$

BV_1 and BV in the first and second equations, respectively, are the results of assignment computations. By the time a bitvector becomes the sole remaining argument, the computation has been completely evaluated. The first equation will replace any pending update event to the same net with an update containing the current value of the assignment right hand side computation, BV_1. Gordon refers to this idea as *cancelling*. The second equation is an *otherwise* equation [2] that is only applied if the first equation does not match (there is no pending continuousUpdateEvent).

4.4 Variable Lookup and Updating

Net/variable lookup and updating is performed by rules. The reason for this is that the lookup and updating of variables can affect the state of other concurrent processes. As well as a theoretical argument that this constructs must be rules because they affect the output of the program, there is a practical argument: rewriting logic supporting tools such as Maude [2] (which is used for our definition) are able to search the non-deterministic state space of a running Verilog program, but it only uses the rules of the definition, not the equations, to search non-deterministic choices.

The value of a given net/variable is simply found in the environment. We only show a sample lookup rule here:

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(\mathsf{exp}(N, X) \curvearrowright K)\ \mathit{PS})\ \mathsf{env}(X \leftarrow \mathit{BV}, \mathit{Env}) \\ \rightarrow \mathsf{activeProcesses}(\mathsf{k}(\mathit{BV} \curvearrowright K)\ \mathit{PS})\ \mathsf{env}(X \leftarrow \mathit{BV}, \mathit{Env}).\end{array}$$

BV is a bitvector, *PS* is the remaining active processes, *K* is the rest of the computation, *X* is the variable or net name, and *Env* is the rest of the environment. The value sizing has been removed for simplicity (complete definition available at [9]). There are also rules for variable or net slice lookups, and lookups of both varieties for continuous (wire assign) processes. These are similar and thus omitted.

Net/variable updating occurs when an update event executes. Update events are generated by the assignments, as explained in the previous subsection. Update events must update the environment, wake up any process that is currently waiting as a listening event, and alert any monitor event that the value has changed:

$$\begin{array}{l}\mathsf{updateEvents}(\mathsf{updateEventList}(\mathsf{updateEvent}(X,\mathit{BV}_1); \mathit{EL}, K)\mathit{ES}_1) \\ \mathsf{env}(X \leftarrow \mathit{BV}_2, \mathit{Env}) \\ \mathsf{listeningEvents}(\mathit{ES}_2) \\ \mathsf{monitorEvents}(\mathit{ES}_2) \\ \rightarrow \mathsf{updateEvents}(\mathsf{updateEventList}(\mathit{EL}, K)\mathit{ES}_1) \\ \mathsf{env}(X \leftarrow \mathit{BV}_1, \mathit{Env}) \\ \mathsf{listeningEvents}(\mathsf{sense}(X, \mathit{BV}_2, \mathit{BV}_1, \mathit{ES}_2)) \\ \mathsf{monitorEvents}(\mathsf{sense}(X, \mathit{BV}_2, \mathit{BV}_1, \mathit{ES}_3))\end{array}$$

ES's represent sets of events. The updateEventList groups several update events as a list. This allows us to ensure that non-blocking assignments occur in program text order within a procedural block, as well as allowing for an easy and clear representation of concatenation-form assignments (see [9]). The operator sense is responsible for waking up the proper listening events and deciding if any monitor event need execute in the current simulator cycle. It uses the previous and current value of the variable or net that is updated to make its determinations.

The last equation shown here schedules the K from the updateEventList to execute when all update events in the updateEventList have been exhausted. For non-blocking assignments, the term K will be empty, meaning that nothing is scheduled (see Section 4.6).

$$\begin{array}{l}\mathsf{updateEvents}(\mathsf{updateEventList}(\mathsf{empty}, K)\ \mathit{ES}) \\ \mathsf{activeProcesses}(\mathit{PS}) \\ = \mathsf{updateEvents}(\mathit{ES}) \\ \mathsf{activeProcesses}(\mathsf{k}(K)\ \mathit{PS})\end{array}$$

4.5 Delays and Triggers

The semantics for delays and triggers is fairly straightforward. Delays simply move the current active process to the future event set with a simulator time equal to the current time added to the time of the delay, if the delay is non-zero. If the delay is zero, the rest of the active processes is moved to the set of inactive events set. Triggers add the rest of the current active process to the set of listening events. The equations for each of these follow.

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(\mathsf{stmt}(\#\ \mathit{NzN}\ S) \curvearrowright K)\ \mathit{PS}) \\ \mathsf{futureEvents}(\mathit{EL}) \\ \mathsf{time}(N) \\ = \mathsf{activeProcesses}(\mathit{PS}) \\ \mathsf{futureEvents}(\mathsf{futureEvent}(N + \mathit{NzN}, \mathsf{stmt}(S) \curvearrowright K); \mathit{EL}) \\ \mathsf{time}(N)\end{array}$$

Here, in the equation for non-zero delays, *NzN* is a non-zero natural number, while *N* is any natural number. N is the current time, while *NzN* is the delay. As expected, the rest of the current process, K, is added to the future event set, as well as the delayed statement S. The first argument to the futureEvent operator is the simulator cycle in which the event should be scheduled to run as an active process.

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(\mathsf{stmt}(\#\ \mathbf{0}\ S) \curvearrowright K)\ \mathit{PS}) \\ \mathsf{inactiveEvents}(\mathit{EL}) \\ = \mathsf{activeProcesses}(\mathit{PS}) \\ \mathsf{inactiveEvents}(\mathsf{inactiveEvent}(\mathsf{stmt}(S) \curvearrowright K); \mathit{EL})\end{array}$$

This equation for zero delayed statements above moves the rest of the inactive process to the inactive events set.

$$\begin{array}{l}\mathsf{activeProcesses}(\mathsf{k}(\mathsf{stmt}(@(\mathit{SL})\ S) \curvearrowright K)\ \mathit{PS}) \\ \mathsf{listeningEvents}(\mathit{EL}) \\ = \mathsf{activeProcesses}(\mathit{PS}) \\ \mathsf{listeningEvents}(\mathsf{listeningEvent}(\mathit{SL}, \mathsf{stmt}(S) \curvearrowright K); \mathit{EL})\end{array}$$

The equation above adds the trigger process to the listening events set. *SL* is the sensitivity list, maintained as the first argument to listeningEvent so that sense can decide which listening events must be scheduled at an update event.

4.6 Process/Event Scheduling

We next present some rules for scheduling the main simulator loop. The general ordering of events was given in Section 2.4. The general idea is to continue with the next set of events in the list when all the previous sets are empty.

Active processes and update events can run at any time. The inactive events, if any, are activated as soon as the active processes and update events are exhausted:

$$\begin{array}{l} \mathsf{activeProcesses}(\mathsf{empty}) \\ \mathsf{updateEvents}(\mathsf{empty}) \\ \mathsf{inactiveEvents}(\mathit{NES}) \\ \rightarrow \mathsf{activeProcesses}(\mathsf{activate}(\mathit{NES})) \\ \mathsf{updateEvents}(\mathsf{empty}) \\ \mathsf{inactiveEvents}(\mathsf{empty}) \end{array}$$

The operator activate schedules each individual inactive event as an active process with its own k operator. Here the term *NES* is specifically a *non-empty* set of events.

If there are no inactive events when all the active processes and update events are exhausted, then the non-blocking assign update events are activated simultaneously by moving them to the update event set:

$$\begin{array}{l} \mathsf{activeProcesses}(\mathsf{empty}) \\ \mathsf{updateEvents}(\mathsf{empty}) \\ \mathsf{inactiveEvents}(\mathsf{empty}) \\ \mathsf{nonBlockingAssignUpdate}(\mathit{NEL}) \\ \rightarrow \mathsf{activeProcesses}(\mathsf{empty})) \\ \mathsf{updateEvents}(\mathsf{updateEventList}(\mathit{NEL}, \mathsf{empty})) \\ \mathsf{inactiveEvents}(\mathsf{empty}) \\ \mathsf{nonBlockingAssignUpdateEvents}(\mathsf{empty}) \end{array}$$

The variable *NEL* denotes a non-empty list of events. The events in the non-blocking assign update event set are added to one updateEventList. The continuation argument is empty, as there is nothing to continue after a non-blocking assignment changes the state of the program, as mentioned in Section 4.4. The rest of the scheduling rules are omitted; they may be found in the full definition [9].

5 Using the Semantics

The first use of the semantics, as a tool, is running Verilog programs within Maude [2]. The Maude rewrite command simply chooses some fair deterministic order for applying rules. If the rewrite command is used with the Verilog semantics and a Verilog program, it is similar to a simulator for Verilog. What gives us extra power, however, is the Maude search command. This command, as mentioned briefly earlier, allows for searching all possible non-deterministic choices during the rewriting process. The upshot of this is that if there is *visible* non-determinism in a Verilog program it will become manifest as multiple possible outputs from the program. It is also possible to apply the Maude linear temporal logic model checker to search for possible safety and liveness violations. The only caveat to these uses is that the Verilog programs must be of fairly small size (less than approximately 40,000 lines of code).

```
module propagation_loop;

  reg [15:0] x;

  initial
  begin
    x  = 0;
    x <= 2;
    #10 $display("x = %d", x);
    $finish;
  end

  always @(x[0])
  begin
    x = x + 1;
  end

endmodule
```

Figure 6. Propagation Loop Example

Displaying the possible outputs of non-deterministic programs using the search command allows for verifying the results of simulators or other formal tools. For instance, the example in Figure 6 shows a Verilog program that terminates with `x = 3` in VCS, but in Iverilog v.092 produces an infinite loop. Running the semantics tells us that `x = 3` is a possible solution, but an infinite loop is not. This was submitted to the Iverilog community, who have agreed that this is a bug. Several other examples of differing output between the two exist [9]. Some of these examples can be determined to *both* be correct with respect to the semantics as we have defined them, meaning that the output of each was listed as a possible output using search.

6 Related Work

In [6], Michael Gordon presents a formal semantics for a simplified version of Verilog called V. V does not deal with many of the features of Verilog that our definition does (such as value sizing). Additionally, it uses new terminology rather than that of the standard. While the syntax described in the paper is formal, the semantics, as presented, are primarily in English language form. Additionally, the semantics presented is not executable, making it more difficult to ask questions about what output a given program should produce.

Gordon Pace and Jifeng He present a brief formal semantics of Verilog in [14]. While completely formal, the definition they present does not cover several major features of Verilog, such as non-blocking assignments Additionally, their semantics is not executable.

In [18], Hisashi Sasaki presents a semantics for Verilog in terms of abstract state machines. His definition is not executable, and it does not capture the inherent nondeterminism of Verilog, which we feel is one of the most important issues to understanding Verilog.

In [24], Huibiao Zhu, Jifeng He, and Jonathan Bowen present an algebraic semantics of Verilog, which they use to derive a denotational semantics. Their semantics cover a smaller subset of the language than He's earlier work in [14]: not even net assignments are covered.

Of the definitions that we have found, ours is the closest to covering the whole of the Verilog language and its inherent nondeterminism. Only a few small features such as tasks are not in the current definition, and we intend to add them. Additionally, only our semantics is executable, allowing for experimentation with Verilog programs.

7 Conclusions

We introduced a new *executable* formal semantics for Verilog. Formal definitions were presented for several of the key constructs of Verilog. We believe that our definition can be useful both for clearing up misunderstandings about the standard as well as a starting point for discussion on exactly what the standard should entail, *e.g.,* should net assignments be treated deterministically, as presented in Section 4.3, or non-deterministically (as an **`always`** block with one blocking assignment) as the standard seems to imply? Since our definition is executable, asking questions about what a certain Verilog program should output is far easier. The full definition is available at [9].

References

[1] P. Borovansky, C. Kirchner, H. Kirchner, and P. E. Moreau. ELAN from a Rewriting Logic Point of View. *Theoretical Computer Science*, 285(2):155–185, 2002.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Springer, 2007.

[3] T. F. Șerbănuță, G. Roșu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.

[4] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[5] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[6] M. J. C. Gordon. The semantic challenge of Verilog HDL. In *Logic in Computer Science (LICS'95)*, pages 136–145, 1995.

[7] IEEE Standard for Verilog Hardware Description Language. IEEE Std 1364-2005, 2005.

[8] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[9] P. Meredith, M. Katelman, J. Meseguer, and G. Roșu. Formal executable semantics of Verilog (webpage), 2010. `http://fsl.cs.uiuc.edu/k/languages/verilog`.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[11] J. Meseguer and G. Roșu. Rewriting logic semantics: From language specifications to formal analysis tools. In *International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *LNAI*, pages 1–44. Springer, 2004.

[12] J. Meseguer and G. Roșu. The rewriting logic semantics project. *Theoretical Comp. Science*, 373(3):213–237, 2007.

[13] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Compiler Construction (CC'03)*, pages 61–76, 2003.

[14] G. J. Pace and J. He. Formal reasoning with verilog hdl. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, 1998.

[15] G. D. Plotkin. The origins of structural operational semantics. *Logic and Algebraic Programming*, 60-61:3–15, 2004.

[16] G. D. Plotkin. A structural approach to operational semantics. *Logic and Algebraic Programming*, 60-61:17–139, 2004. Previously published as Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[17] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 2010. DOI: 10.1016/j.jlap.2010.03.012.

[18] H. Sasaki. A formal semantics for verilog-vhdl simulation interoperability by abstract state machine. In *Design, automation and test in Europe (DATE'99)*, pages 73–78. ACM, 1999.

[19] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Symposium on Computers and Automata*, volume XXI, pages 19–46, Brooklyn, N.Y., 1971. Polytechnic Press.

[20] Synopsys. VCS. `http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx`.

[21] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[22] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.

[23] S. Williams. Icarus verilog. `http://www.icarus.com/eda/verilog/`.

[24] H. Zhu, J. He, and J. P. Bowen. From algebraic semantics to denotational semantics for verilog. In *International Conference on Engineering Complex Computer Systems (ICECCS'06)*, pages 139–151, 2006.