Putting Operational Techniques to the Test: A Syntactic Theory for Behavioral Verilog

John Fiskio-Lasseter and Amr Sabry

Department of Computer & Information Science
University of Oregon
Eugene, OR USA

Abstract

We present a syntactic theory for the behavioral subset of the Verilog Hardware Description Language. Due to the complexity of the language, the construction of this theory represents a serious test of the suitability of syntactic operational techniques for reasoning about industrial languages. Overall, we have found that these techniques are rather robust but with a few caveats. Our theory formalizes the simulation cycle explicitly, exposes a number of ambiguities and inconsistencies in the language reference manual (LRM), and is the most accurate known description of this subset of Verilog, with respect to the LRM. The syntactic theory has been used to automatically derive a simulator for Verilog.

1 Introduction

Programming calculi, which concentrate on a small set of constructs that capture the "essence" of a language, commonly come equipped with syntactic theories that explain, in intuitive yet formal terms, the evaluation and optimization of programs. In principle, then, the development of such theories for full-fledged industrial programming languages appears not only possible but even straightforward. Verilog, an industrial hardware-description language, is a candidate language for which a syntactic theory would be much welcome. First, Verilog has an incredibly rich set of constructs, and the behavior of many of these is not well understood. Its semantic formalization is a recognized challenge [10], yet there exists no complete semantics for the language, operational or otherwise. Hence, developing a syntactic theory for Verilog represents a good test of the expressiveness and scalability of operational techniques. Second, the informal semantics of Verilog in the standard

[1] This work is supported by the National Science Foundation under the Title "Career: Controlling Space Properties of Higher-Order Typed Programs" (grant number CCR-9733088).

language reference manual (LRM) [13] is already given in an operational style, but simulators that implement the informal semantics differ in subtle ways. A syntactic theory can therefore identify many of the inconsistencies in the informal semantics and yield a correct-by-construction standard simulator that defines *and implements* the semantics.

This paper presents the development of a syntactic theory for behavioral Verilog, which clarifies and exposes a number of ambiguities and inconsistencies in the LRM. With respect to the LRM, the resulting formal semantics for the language is the most accurate of any work to date. From a semantic point of view, the development steps themselves are also interesting:

- First, the development of a syntactic theory for Verilog essentially requires one to encode the state of the simulator in the syntax of terms. Part of this encoding is nowadays standard: for example, the idea of encoding the store as a sequence of declarations. Encoding of other constructs required new innovations, while still others (*e.g.* continuous assignments) have resisted encoding outright.
- Second, the complexity of the simulation cycle motivates the introduction of "state-space syntax" to conveniently express the state of the simulator. This situation is not uncommon: for example, one might introduce a let-notation to conveniently express the axioms of the call-by-need calculus [1,2], even though it is technically not necessary. Unlike the situation with the let-notation which has a well-known expansion in the core language, our state-space syntax cannot be naively expanded to the core language without a proof obligation. This rigorous treatment of state-space syntax clarifies its role in the development of syntactic theories.
- Finally, the complexity of the syntactic theory is such that only the most trivial examples can be simulated by hand. Xiao *et al* have developed a tool that automatically generates a rewriting-based interpreter given a semantic description [21]. We have used this tool to implement a correct-by-construction simulator for Verilog.

The remainder of the paper presents our theory in detail. The next section reviews related work, and section 3 introduces the syntax and informal semantics of our behavioral subset of Verilog. Section 4 introduces the state-space syntax in which the syntactic theory is expressed and develops the axioms needed to express the possible behaviors of Verilog simulators. Following this (section 5), we argue that state-space syntax is not mere notation; its introduction entails a proof obligation. Lastly, section 6 discusses the automatic generation of a complete simulator from the axioms.

2 Related Work

The style of semantic specification based on axioms and reductions was pioneered by Plotkin in 1975 [16]. The original work explained the semantics of two small pure functional languages: a call-by-value one and a call-by-name one. Since then, the specification technique has been successfully applied to explain all sorts of language constructs including assignments, jumps, and continuation-based control operators [8,14,17,19]. Our current work extends this application to languages with threads, timing and event controls, and hardware-specific constructs like edges and non-blocking assignments.

Within industry, there are two problems that drive much of the research into the semantics of Verilog and the related language VHDL: synthesis of low-level designs from high-level ones and interoperability between Verilog and VHDL designs. There are operational semantic descriptions for Verilog and/or VHDL based on timed finite automata [5,6], higher order logic [20], abstract state machines [18], and various calculi [11,15]. All of these define semantics for restricted subsets of the language, and, with few exceptions [18], the behavior of many constructs is idealized and not accurate with respect to the requirements of the LRM.

Axiomatic specification of hardware description languages is relatively new, although Hua and Zhang [12] do describe an axiomatic semantics for a subset of the experimental *Iowa Logic Specification Language*.

Many of the subtleties of the Verilog language represented in our semantics are described informally in Gordon's "Semantic Challenge" paper [10]. For the most part, we follow his description, but for certain constructs, we have found it necessary to depart from his interpretation.

An earlier version of this work is available as a technical report [9]. A second report, detailing the most recent state of our work, is in preparation.

3 Behavioral Verilog

In the full language, a Verilog program is a collection of one or more modules. Modules bear an intuitive resemblance to classes from object-oriented languages, but their bodies consist primarily of a collection of procedural statements (*threads*) and instantiations of other modules. A module body can also model wiring connections in the form of `assign` statements (*continuous assignments*), or with a list of parameters (*port list*), to which variables are supplied as arguments for each instantiation. All threads and continuous assignments, in each instantiated module, are considered to run concurrently. In every Verilog program, one of the modules is declared without a port list and is considered to be at the *top-level*. All other declared modules are simulated in terms of this top-level behavior, according to how they are instantiated.

Here we limit our formal description to a core subset of Verilog, consisting only of behavioral elements. These are the constructs, drawn from ordinary procedural languages, that are relevant to simulating designs at a high-level of abstraction. We consider only Verilog programs consisting of a single top-level module, excluding continuous assignments, variables of `wire` type and module instantiations.

3.1 Syntax and Informal Semantics

The syntax of programs, threads, statements, expressions, and timing controls is:

$$
\begin{array}{rrl}
\textit{(programs)} & P_v ::= & \texttt{module}\ m\texttt{;}\ d_v\ t_v{}^*\ \texttt{endmodule} \\
\textit{(declarations)} & d_v ::= & \ \mid \texttt{reg}\ v\texttt{;}\ d_v \mid \texttt{event}\ u\texttt{;}\ d_v \\
\textit{(threads)} & t_v ::= & \texttt{always}\ s \mid \texttt{initial}\ s \\
\textit{(statements)} & s ::= & \ \mid v\texttt{=}e\texttt{;} \mid v\texttt{=}c\ e\texttt{;} \mid v\texttt{<=}e\texttt{;} \mid v\texttt{<=}c\ e\texttt{;} \mid \texttt{->}u\texttt{;} \\
 & & \mid \texttt{forever}\ s \mid \texttt{while}\ (e)\ s \mid c\ s \mid s\texttt{;}\ s \\
 & & \mid \texttt{if}\ (e)\ s \mid \texttt{if}\ (e)\ s\ \texttt{else}\ s \\
\textit{(timing control)} & c ::= & \texttt{@}(g) \mid \texttt{\#}e \\
\textit{(guards)} & g ::= & v \mid u \mid \texttt{posedge}\ v \mid \texttt{negedge}\ v \mid g\ \texttt{or}\ g \\
\textit{(expressions)} & e ::= & v \mid n \mid uop\ e \mid e\ bop\ e \mid e\texttt{?}e\texttt{:}e
\end{array}
$$

The value contained in a register is either a number, `x` (*unknown value*), or `z` (*high impedance*). An event declaration introduces a name that can be used to communicate among threads. Such a variable can be *triggered* (`->` u) but holds no data. We use v to range over `reg` variables, u to range over `event` variables, and n to range over $\{\mathbb{Z} \bigcup \{\texttt{x},\texttt{z}\}\}$. We also use *uop* and *bop* to range over the unary and binary operators in Verilog.

Expressions and some of the statements should look familiar to C programmers. Indeed the semantics of familiar statements like conditionals and loops is the expected one.

The remaining constructs are specific to Verilog and are described in terms of the *simulation cycle* of programs [13, §5]. Initially all threads in the program are active and can be executed concurrently. The statement in an `initial`-thread is run exactly once whereas the statement in an `always`-thread is repeatedly evaluated. Within each active thread, the execution of statements proceeds sequentially (we ignore Verilog's `fork/join` construct) until the next timing control.

In addition to ordinary assignments of the form v`=`e (*blocking assignment*), Verilog also supports the unusual *non-blocking assignment*, of the form v`<=`e. The effect of this assignment is to evaluate e immediately, resulting in some value, n. Unlike a blocking assignment, the variable v is not assigned the value n immediately. Rather, all such updates are *scheduled* to execute at the end of the current simulation cycle (in the order in which the original non-blocking assignments occur). Meanwhile, execution of the rest of the thread proceeds without interruption, using the value of v before the non-blocking assignment. Note that a blocking assignment to v later in the thread body will be overwritten by the non-blocking update at the end of the cycle.

If the execution of a thread encounters an *event control* (`@`(g)), it is immediately blocked until the specified event, g, occurs. The possible events are either explicit signals (of the form `->`u), or changes in the value of a variable (*edges*). Guards of the form `@(posedge` v`)`, `@(negedge` v`)`, and `@`(v) will fire on rising edges, falling edges, or any edge (respectively), as soon as v changes value; those of the form `@`(u) will fire as soon as the corresponding event trigger occurs. A *guarded* thread can become active during any phase of the cycle, depending on when its guarded-upon event happens.

With an explicit *delay* `#`e, execution of a thread is suspended for e cycles. At the end of a cycle, (*i.e.*, after the nonblocking updates), the simulation clock advances to the time of the earliest scheduled delayed thread, which is then made active. The peculiar case of a thread blocked by `#0` is also allowed. Such threads, called *inactive* or *δ-delayed*, are suspended until a later point in the current cycle: after all the other active threads have finished or blocked, but before any pending non-blocking updates.

Both forms of timing control may also be used in *intra-assignment delays*, of the form v`=`$c\ e$ or v`<=`$c\ e$. In blocking assignments, e is immediately evaluated to get a number, n. The thread is then blocked according to the evaluation of c: when it is released, v will immediately be updated with the value n. Non-blocking assignment with delay is similar, except that execution of the thread continues uninterrupted, with a non-blocking assignment of n to v scheduled according to the specified timing control.

3.2 Behavioral vs. *Structural Verilog*

Although it is not part of our theory, Verilog also includes a set of low-level constructs which model hardware elements directly. This *structural* subset includes continuous assignments and port connections, but also includes constructs for modelling elements as primitive as transistors. It is at least as rich as the behavioral subset, but the behavior of the constructs and their applications are fundamentally different. In particular, the structural subset contains no notion of procedural control.

Consider, for example, a behavioral model of an ordinary SR latch:

```
module beh_sr_test;
  reg q, r, s;
  always @(q or r or s)
    if (r) q = 0 else q = s | q;
      ⋮
endmodule
```

A similar (although not identical) structural implementation might be:

```
module struc_sr_test;
  wire q; reg r, s;
  assign q = ~(r |~(s | q));
      ⋮
endmodule
```

The main semantic difference between the two implementations is that the blocking assignment in the behavioral implementation happens only when this statement is reached during the flow of control. By contrast the `assign` is considered to execute constantly: the notion of sequential control does not apply. At every moment, the value of q is the value at that moment of the right-hand side.

As an aside, our semantics can in fact handle many Verilog programs with module instantiations, `wire` variables and continuous assignment. In many cases, such elements can be eliminated by performing an inlining transformation [10]. Such a transformation works by using α-renaming to declare all variables at top-level and then applying expression substitution to collapse all modules into the top-level and eliminate all `module` instantiations, port connections, and `wire` variables. Because of the presence of run-time variable allocation, such a transformation would be unsound in most languages, of course. Felicitously, Verilog lacks any form of recursion or even dynamic memory allocation. All variables are static, and hence it is always possible to determine the space requirements of a program at compile-time.

4 Syntactic Theory

We will explain the semantics of Verilog using a syntactic theory that consists of axioms or rewriting rules that relate Verilog programs to other Verilog programs. In doing so, we must represent not only sequential control and memory, but also the state of the discrete event queue as the simulation cycle progresses. Although it is possible to use Verilog syntax during the development of the theory, we find it much more convenient to use a syntax in which we can cleanly represent the state-space.

4.1 State-Space Syntax

The state-space language, which we will call $\mathtt{V}_s$, uses the same definitions of statements, declarations, expressions, and timing controls as those defined for Verilog but makes the store and the state of each thread explicit:

$$\begin{array}{lrcl}
\textit{(programs)} & P_s & ::= & d_v\ \mathtt{m}_1\{\phi\}\mathtt{m}_2\{\phi^+\}\ t_s{}^*\ \mathtt{nbu}\{\theta\} \\
\textit{(elementary store)} & \phi & ::= & \mid v\texttt{=}n\texttt{;}\ \phi \\
\textit{(augmented store)} & \phi^+ & ::= & \phi \mid \phi^+\ \texttt{->}u\texttt{;} \\
\textit{(nbu elements)} & \theta & ::= & \mid v\texttt{<=}n\texttt{;}\ \theta \\
\textit{(threads)} & t_s & ::= & \mathtt{at}\{s\} \mid \mathtt{zd}\{s\} \mid \mathtt{fg}\{\texttt{@}(g)\ s\} \\
 & & & \mid \mathtt{fd}\{\texttt{\#}n\ s\} \qquad \bullet\ n > 0
\end{array}$$

Each thread is annotated with a tag describing whether it is active (`at`), 0-delayed (`zd`), future guarded (`fg`; *i.e.*, suspended at a guard), or future delayed (`fd`). The scheduled non-blocking updates are also explicitly described using the tag `nbu`. Finally the state-space syntax includes not one, but two copies of the store ($\mathtt{m}_1$ and $\mathtt{m}_2$). The set ϕ must consist of exactly one assignment to a `reg` variable v_i, for each declared v_i in P_s. If, for example, P_s contains k declared `reg` variables, ϕ must be of the form $v_1\texttt{=}n_1\texttt{;}\ \ldots\ v_k\texttt{=}n_k\texttt{;}$.

Each axiom in the theory relates two syntactic constructs by the equivalence relation $\stackrel{ax}{=}$. Although this relation is symmetric, each axiom is in practice used to rewrite the left hand side of the equation to the form on the right. We identify terms modulo bound variables, and we assume that free and bound variables do not interfere in definitions or theorems. In other words, we work with the quotient under α-equivalence [3].

Since the axioms are going to be expressed using the state-space syntax, we must transform a Verilog program to $\mathtt{V}_s$ before evaluating it. This step is straightforward: initially the store contents are un-initialized, there are no pending updates, and every thread is marked active. Hence the mapping from Verilog to state-space is:

$$
\begin{aligned}
&S : P_v \longrightarrow P_s\\
&S_d : d_v \longrightarrow d_v\\
&S_t : t_v \longrightarrow t_s
\end{aligned}
$$

$$
\begin{aligned}
&S(\texttt{module m;}\ \ d_v\ {t_v}^n\ \ \texttt{endmodule})\\
&\qquad\qquad = S_d(d_v)\ (S_t(t_v))^n\ \ \texttt{nbu}\{\ \}\\
&S_d(\texttt{reg}\ v_1,\ldots,v_k\texttt{;}\ \ \texttt{event}\ u_1,\ldots,u_{k'}\texttt{;})\\
&\qquad\qquad = \texttt{reg}\ v_1,\ldots,v_k\texttt{;}\ \ \texttt{event}\ u_1,\ldots,u_{k'}\texttt{;}\\
&\qquad\qquad\quad\ \texttt{m}_1\{v_1\texttt{=x;}\ \ldots v_k\texttt{=x;}\}\ \ \texttt{m}_2\{v_1\texttt{=x;}\ \ldots v_k\texttt{=x;}\}\\
&S_t(\texttt{always}\ s)\ = \texttt{at}\{\texttt{forever}\ s\}\\
&S_t(\texttt{initial}\ s) = \texttt{at}\{s\}
\end{aligned}
$$

The $\mathtt{x}$ on the right hand side of each assignment in $\mathtt{m}_1$ and $\mathtt{m}_2$ corresponds to the "unknown value" in Verilog [13, §3.2.2]. For a Verilog procedural block (thread) t_v, ${t_v}^n$ denotes the occurence of n consecutive (not necessarily identical) t_v.

4.2 Axioms for Simple Constructs

As with most languages, the ordinary procedural constructs in Verilog are easy to describe in an axiomatic fashion, because they have no global effects. They are described here by the following axioms which can be applied in any context:

(*ax_uop*)	$uop\ n \stackrel{ax}{=} \delta_1(uop, n)$	• if defined
(*ax_bop*)	$n_1\ bop\ n_2 \stackrel{ax}{=} \delta_2(bop, n_1, n_2)$	• if defined
(*ax_cond_true*)	$n\texttt{?}e_1\texttt{:}e_2 \stackrel{ax}{=} e_1$	• if $n \notin \{0, \mathtt{x}, \mathtt{z}\}$
(*ax_cond_unk*)	$n\texttt{?}e_1\texttt{:}e_2 \stackrel{ax}{=} \delta_3(e_1, e_2)$	• if $n \in \{\mathtt{x}, \mathtt{z}\}$
(*ax_cond_false*)	$0\texttt{?}e_1\texttt{:}e_2 \stackrel{ax}{=} e_2$	
(*ax_if_true*)	$\texttt{if}\ (n)\ s \stackrel{ax}{=} s$	• if $n \notin \{0, \mathtt{x}, \mathtt{z}\}$
(*ax_if_false*)	$\texttt{if}\ (n)\ s \stackrel{ax}{=} skip$	• if $n \in \{0, \mathtt{x}, \mathtt{z}\}$
(*ax_ifelse_true*)	$\texttt{if}\ (n)\ s_1\ \texttt{else}\ s_2 \stackrel{ax}{=} s_1$	• if $n \notin \{0, \mathtt{x}, \mathtt{z}\}$
(*ax_ifelse_false*)	$\texttt{if}\ (n)\ s_1\ \texttt{else}\ s_2 \stackrel{ax}{=} s_2$	• if $n \in \{0, \mathtt{x}, \mathtt{z}\}$

$(ax\_forever)$	`forever` $s \stackrel{ax}{=}$ `while` (1) s
(ax_while)	`while` (e) $s \stackrel{ax}{=}$ `if`(e) `begin` s; `while` (e) s `end`
(ax_cba)	§4.5, below.

Following tradition, we use the functions δ_1 and δ_2 to abstract the semantics of the built-in unary and binary operators [16]. Similarly, δ_3 abstracts the rather complicated evaluation of ambiguous conditional expressions [13, §4.1.13].

4.3 Evaluation Contexts

The remaining constructs included in our semantics require a little more sophistication. These have global effects—either on the store, the simulation cycle, or both. They must therefore be performed in a certain order, and in some cases, only at certain points in the simulation cycle. Such constraints can be specified syntactically using *evaluation contexts* [8].

$$
\begin{array}{lrcl}
(\mathit{expressions}) & \mathcal{E} & ::= & [\,] \mid uop\ \mathcal{E} \mid \mathcal{E}\ bop\ e \mid n\ bop\ \mathcal{E} \mid \mathcal{E}\texttt{?}e\texttt{:}e \\
(\mathit{statements}) & \mathcal{S} & ::= & \texttt{if}\ (\mathcal{E})\ s \mid \texttt{if}\ (\mathcal{E})\ s\ \texttt{else}\ s \mid v\texttt{=}\mathcal{E} \mid v\texttt{<=}\mathcal{E} \\
 & & & \mid v\texttt{=}c\ \mathcal{E} \mid v\texttt{<=}c\ \mathcal{E} \mid \mathcal{S}; s \mid \texttt{\#}\mathcal{E}\ s \\
(\mathit{guards}) & \mathcal{G} & ::= & [\,] \mid \mathcal{G}\ \texttt{or}\ g \mid g\ \texttt{or}\ \mathcal{G} \\
(\mathit{memory\ lookup}) & \mathcal{M} & ::= & [\,] \mid \mathcal{M}\ \ v\texttt{=}n\texttt{;} \mid \mathcal{M}\ \ \texttt{->}u\texttt{;} \\
(\mathit{thread\ selection}) & \mathcal{T} & ::= & [\,] \mid t_s\ \mathcal{T} \mid \mathcal{T}\ t_s \\
(\mathit{programs}) & \mathcal{P} & ::= & d_v\ \texttt{m}_1\{\phi\}\texttt{m}_2\{\phi^+\}\ \mathcal{T}\ \texttt{nbu}\{\theta\} \\
(\mathit{programs}) & \mathcal{P}' & ::= & d_v\ [\,]\ \texttt{nbu}\{\theta\} \\
(\mathit{programs}) & \mathcal{P}'' & ::= & d_v\ \texttt{m}_1\{\phi\}\ [\,]\ \texttt{nbu}\{\theta\}
\end{array}
$$

The contexts for expressions and statements have fairly ordinary definitions. Intra-assignment timing controls (statements of the form v`=`$c\ e$ or v`<=`$c\ e$) are similarly uneventful, although the reader should note that we require evaluation of the right-hand side before that of the timing control [13, §9.7.6]. The contexts for guards, memory lookup, and thread selection allow for nondeterministic ordering.

The three program contexts are used primarily to shorten many of our axioms. They are used almost everywhere, but the most straightforward applications are in a pair of "housekeeping" axioms, which are useful for garbage collection and rearranging the source code ordering of threads:

$$
\begin{array}{ll}
(ax\_tgc) & \mathcal{P}[\texttt{at}\{\}] \stackrel{ax}{=} \mathcal{P}[\,] \\
(ax\_ord) & \mathcal{P}[t_{s_1}\ t_{s_2}] \stackrel{ax}{=} \mathcal{P}[t_{s_2}\ t_{s_1}]
\end{array}
$$

The purpose of (ax_tgc) should be clear. Axiom (ax_ord) allows us to prove the equivalence of two programs that differ only in the source-code ordering of their threads. This irrelevance of thread ordering is also implied by the nondeterministic selection of any thread in a context $\mathcal{T}$.

4.4 Assignments Revisited

To axiomatize assignments, we extend a technique introduced by Boehm [4] that uses a sequential block at the beginning of the program to represent the store. This block consists of a sequence of assignments to constants (representing the most recently assigned values), one for each declared variable in the program.

Because the evaluation of Verilog programs needs to detect edges (changes in the store), we keep *two* copies of the store, $\mathtt{m}_1$ and $\mathtt{m}_2$. In this scheme, $\mathtt{m}_2$ is used to represent the most recently assigned value to a variable. The value held by that variable just before the assignment is kept in $\mathtt{m}_1$. The difference in assignments corresponds to the edge.

We also use the store to keep track of event triggers: when one occurs, we just add it to $\mathtt{m}_2$, deleting it entirely after its effects have been applied.

With these techniques in place, we have four axioms that involve the store:

$$\begin{array}{ll}
(\mathit{ax_block_assign}) & \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_1]\}\ \mathcal{T}[\mathtt{at}\{v\texttt{=}n_2\texttt{;}\ s\}]\ ] \\
 & \quad \stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_2]\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ] \\
 & \\
(\mathit{ax_lookup}) & \mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n]\}\ \mathcal{T}[\mathtt{at}\{\mathcal{S}[v]\}]\ ] \\
 & \quad \stackrel{ax}{=} \mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n]\}\ \mathcal{T}[\mathtt{at}\{\mathcal{S}[n]\}]\ ] \\
 & \\
(\mathit{ax_event_trig}) & \mathcal{P}''[\ \mathtt{m}_2\{\phi^+\}\ \mathcal{T}[\mathtt{at}\{\texttt{->}u\texttt{;}\ s\}]\ ] \\
 & \quad \stackrel{ax}{=} \mathcal{P}''[\ \mathtt{m}_2\{\phi^+\ \ \texttt{->}u\texttt{;}\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ] \\
 & \\
(\mathit{ax_nb_assign}) & d_v\ \mathtt{m}_1\{\phi\}\ \mathtt{m}_2\{\phi^+\}\ \mathcal{T}[\mathtt{at}\{v\texttt{<=}n\texttt{;}s\}]\ \mathtt{nbu}\{\theta\} \\
 & \quad \stackrel{ax}{=} d_v\ \mathtt{m}_1\{\phi\}\ \mathtt{m}_2\{\phi^+\}\ \mathcal{T}[\mathtt{at}\{s\}]\ \mathtt{nbu}\{\theta\ \ v\texttt{<=}n\texttt{;}\}
\end{array}$$

The requirement in (*ax_block_assign*) that v be the same in both $\mathtt{m}_1$ and $\mathtt{m}_2$ (*i.e.*, no edges are present on v) arises from the task of guaranteeing correct event control release. In particular, it ensures that we can only cancel an edge under certain conditions (section 4.7). On the other hand, the consistency requirement on v is relaxed for (*ax_lookup*), since expressions cannot cause side-effects in our subset of Verilog. Of course, we would have to change this in a complete implementation.

For both (*ax_event_trig*) and (*ax_nb_assign*), consistency of $\mathtt{m}_1$ and $\mathtt{m}_2$ is not relevant: nonblocking assignments have no immediate effect on the store, and the occurrence of an event trigger does not wipe out other signals already in $\mathtt{m}_2$.

Note further that the usual way of representing the store is to keep a list of *every* assignment, using only the most recent one. In the present work, however, we have chosen to keep only the most recent value. Now for blocking assignments, this is an optimization which is clearly equivalent in behavior. On the other hand, the same optimization cannot be used with non-blocking assignment: every scheduled update must execute. This is so because a non-blocking update can itself trigger an event-controlled thread, even if the update is immediately overwritten.

Finally, note that *the* value of an expression in a certain state is not necessarily well-defined. Axiom (*ax_lookup*) applies only to a single lookup; it does not specify that the entire expression be evaluated. As a result, our semantics supports full interleaving of statements and expression evaluation, as required by the LRM [13, §5.4.1].

4.5 Delay and Intra-Assignment Timing Control

Blocking of a thread on an explicit delay is easy to formalize, involving only a re-labelling of the thread's status in the simulator.

$$
\begin{array}{ll}
(\mathit{ax\_del\_block}) & \mathcal{P}[\mathtt{at}\{\mathtt{\#}n\ s\}] \stackrel{ax}{=} \mathcal{P}[\mathtt{fd}\{\mathtt{\#}n\ s\}] \bullet \text{if n} > 0 \\
(\mathit{ax\_zd\_block}) & \mathcal{P}[\mathtt{at}\{\mathtt{\#0}\ s\}] \stackrel{ax}{=} \mathcal{P}[\mathtt{zd}\{s\}]
\end{array}
$$

It is important to distinguish zero-valued delays (δ-*delay*) from ordinary, nonzero ones. Whereas ordinary delayed threads (`fd`) release during a strictly later simulation cycle, zero-delayed threads (`zd`) must execute in the current one. Note that in (*ax_del_block*), the delay value, `#`n, remains unchanged, whereas the `#0` is deleted in (*ax_zd_block*). This is a somewhat arbitrary choice, dictated by ease of representation in the concrete syntax.

Intra-assignment timing controls are also quite simple:

$$
\begin{array}{lc}
(\mathit{ax\_cba}) & v\mathtt{=}c\ n \stackrel{ax}{=} c\, v = n \\
(\mathit{ax\_cnba}) & \mathcal{P}[\mathtt{at}\{v\mathtt{<=}c\ n\mathtt{;}\ s\}] \stackrel{ax}{=} \mathcal{P}[\mathtt{at}\{s\}\ \mathtt{at}\{c\ v\mathtt{<=}n\}]
\end{array}
$$

The rewrite of a nonblocking assignment with timing control as a new thread corresponds to one of the LRM's more quirky requirements. When two nonblocking assignments are performed in sequence without intra-assignment timing, the order of the scheduled updates is guaranteed. With these timing controls, however, the order of the updates at the scheduled time unit is nondeterministic [13, p.102].

4.6 Synchronization Controls

The behavior of event-control based blocking and release includes a context-sensitive property. In particular, the LRM allows event control constructs to include variables with arbitrarily large bit-widths, but only defines behavior on the low-order bit [13, §9.7.2]. For this purpose, we use the *lsb* function, which returns the value of the least significant bit of its argument.

(*ax_guard_block*)

$$
\begin{array}{l}
\mathcal{P}'[\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathcal{T}[\mathtt{at}\{\mathtt{@}(g)\ s\}]\ ] \\
\quad \stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathcal{T}[\mathtt{fg}\{\mathtt{@}(g)\ s\}]\ ] \\
\qquad \bullet \text{if } \phi_1 = \phi_2
\end{array}
$$

(*ax_posedge_fire*)

$$
\begin{array}{l}
\mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{fg}\{\mathtt{@}(\mathcal{G}[\mathtt{posedge}\ v])\ s\}]\ ] \\
\quad \stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ] \\
\qquad \bullet \text{if } (\mathrm{lsb}(\mathrm{n}_1) \neq 1 \wedge \mathrm{lsb}(\mathrm{n}_2) = 1),\ \text{or } (\mathrm{lsb}(\mathrm{n}_1) = 0 \wedge \mathrm{lsb}(\mathrm{n}_2) \neq 0)
\end{array}
$$

$(ax\_negedge\_fire)$

$$\mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{fg}\{\mathtt{@}(\mathcal{G}[\mathtt{negedge}\ v])\ s\}]\ ]$$
$$\stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ]$$

- if $(\mathrm{lsb}(\mathrm{n}_1) \neq 0 \wedge \mathrm{lsb}(\mathrm{n}_2) = 0)$, or $(\mathrm{lsb}(\mathrm{n}_1) = 1 \wedge \mathrm{lsb}(\mathrm{n}_2) \neq 1)$

(ax_edge_fire)

$$\mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{fg}\{\mathtt{@}(\mathcal{G}[v])\ s\}]\ ]$$
$$\stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v = n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v = n_2]\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ]$$

- if $lsb(n_1) \neq lsb(n_2)$

(ax_event_fire)

$$\mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[\mathtt{->}u]\}\ \mathcal{T}[\mathtt{fg}\{\mathtt{@}(\mathcal{G}[u])\ s\}]\ ]$$
$$\stackrel{ax}{=} \mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[\mathtt{->}u]\}\ \mathcal{T}[\mathtt{at}\{s\}]\ ]$$

The requirement in (ax_guard_block) that both $\mathtt{m}_1$ and $\mathtt{m}_2$ contain an identical ϕ is important. The two copies of memory *must* be consistent before we can allow guards to block. If we allowed any edges to be present in memory, then it would be possible for a guard to release on an event or edge that occurred before the guard itself blocks.

In previous work [9], guarded threads were defined to release as *δ-delayed*, not *active*. This semantics was taken from Gordon's paper [10], and appears to represent the behavior of most leading implementations. The LRM, however, does not specify this detail, and in the present work, a release to *active* (`at`) status is defined. It is thus possible for a thread to block and release within the same *active threads* phase of a cycle. This interpretation seems justified given the presence (in principle) of full interleaving: releasing a guarded thread as `zd` would leave us with no direct support for thread synchronization within a single δ.

4.7 Global Store Axioms

The definitions of assignment, lookup, and signals (4.4) involve only properties of the store itself. Other operations on the store—namely the clearing of existing edges between $\mathtt{m}_1$ and $\mathtt{m}_2$—can only be applied under certain complicated side-conditions. In particular, we must take care never to clear an edge or signal until every blocked thread that can release on this event does so [13, §5.2].

$(ax\_posedge\_clear)$

$$\mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\mathtt{=}n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v\mathtt{=}n_2]\}\ t_s{}^*\ ]$$
$$\stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\mathtt{=}n_2]\}\ \mathtt{m}_2\{\mathcal{M}[v\mathtt{=}n_2]\}\ t_s{}^*\ ]$$

- if both of the following hold :
  - $(lsb(n_1) \neq 1 \wedge lsb(n_2) = 1)$, or $(lsb(n_1) = 0 \wedge lsb(n_2) \neq 0)$
  - there is no thread t_s of the form
    $\mathtt{fg}\{\mathtt{@}(\mathcal{G}[v])\ s\}$, or $\mathtt{fg}\{\mathtt{@}(\mathcal{G}[\mathtt{posedge}\ v])\ s\}$

(*ax_negedge_clear*)

$\mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_2]\}\ {t_s}^*\ ]$

$\stackrel{ax}{=} \mathcal{P}'[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_2]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_2]\}\ {t_s}^*\ ]$

- if both of the following hold :
  - $(lsb(n_1) \neq 0 \wedge lsb(n_2) = 0)$, or $(lsb(n_1) = 1 \wedge lsb(n_2) \neq 1)$
  - there is no thread t_s of the form
    $\mathtt{fg}\{\texttt{@}(\mathcal{G}[v])\ s\}$, or $\mathtt{fg}\{\texttt{@}(\mathcal{G}[\mathtt{negedge}\ v])\ s\}$

(*ax_event_clear*)

$\mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[\texttt{->}u]\}\ {t_s}^*\ ] \stackrel{ax}{=} \mathcal{P}''[\ \mathtt{m}_2\{\mathcal{M}[\ ]\}\ {t_s}^*\ ]$

- if there is no thread t_s of the form
  $\mathtt{fg}\{\texttt{@}(\mathcal{G}[u])\ s\}$

(*ax_bogus_edge*)

$\mathcal{P}''[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_1]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_2]\}\ {t_s}^*\ ]$

$\stackrel{ax}{=} \mathcal{P}''[\ \mathtt{m}_1\{\mathcal{M}[v\texttt{=}n_2]\}\ \mathtt{m}_2\{\mathcal{M}[v\texttt{=}n_2]\}\ {t_s}^*\ ]$

- if $n_1 \neq n_2$, but $lsb(n_1) = lsb(n_2)$

4.8 Global Simulation Axioms

Finally, we must define the advance of the simulation clock itself. There are three ways in which the simulation cycle advances: execution of the δ-delayed threads, execution of the scheduled nonblocking updates, and advancement to the time of the earliest scheduled nonzero delay. Note that both (*ax_zd_release*) and (*ax_del_release*) must be applied to *every* `zd` (/`fd`). Moreover, axioms (*ax_del_release*) and (*ax_nbu_exec*) require non-trivial transformation of the thread bodies, according to the schema specified.

(*ax_zd_release*)

$d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \mathtt{fd}\{s\}^*\ \underline{\mathtt{zd}_1\{s\}\ldots\mathtt{zd}_n\{s\}}\ \mathtt{nbu}\{\theta\}$

$\stackrel{ax}{=} d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \mathtt{fd}\{s\}^*\ \underline{\mathtt{at}_1\{s\}\ldots\mathtt{at}_n\{s\}}\ \mathtt{nbu}\{\theta\}$

- if $\phi_1 = \phi_2$

(*ax_del_release*)

$d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \underline{\mathtt{fd}_1\{s\}\ldots\mathtt{fd}_n\{s\}}\ \mathtt{nbu}\{\ \}$

$\stackrel{ax}{=} d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \underline{\mathtt{at}_1\{s'\}\ldots\mathtt{at}_n\{s'\}}\ \mathtt{nbu}\{\ \}$

- if $\phi_1 = \phi_2$, and
- each $\mathtt{fd}_i\{s\}$ is transformed to a corresponding $\mathtt{at}_i\{s'\}$, according to the schema :

$$\mathtt{fd}\{\texttt{\#}n\ s\} \longrightarrow \mathtt{at}\{\texttt{\#}(n-1)\ s\}, \text{ if } n > 1$$
$$\mathtt{fd}\{\texttt{\#1}\ s\} \longrightarrow \mathtt{at}\{s\}$$

(ax_nbu_exec)

$$d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \mathtt{fd}\{s\}^*\ \underline{\mathtt{nbu}\{\theta\}}$$
$$\stackrel{ax}{=} d_v\ \mathtt{m}_1\{\phi_1\}\ \mathtt{m}_2\{\phi_2\}\ \mathtt{fg}\{s\}^*\ \mathtt{fd}\{\overline{s\}^*\ \ \underline{\mathtt{at}\{\theta'\}\ \mathtt{nbu}\{\ \}}$$

- if $\phi_1 = \phi_2$, and
- each nonblocking assignment in θ is transformed to a blocking assignment in θ', according to the schema :

$$v\mathtt{<=}n \longrightarrow v\mathtt{=}n$$

Here, the consistency requirement on ϕ_1 and ϕ_2, in combination with the axioms in 4.8, ensures that we do not advance the clock when there are still blocked threads that should release and execute first.

5 Formalizing the Relationship Between Verilog and State-Space Syntax

We have proposed a number of axioms to explain the semantics of Verilog. These axioms are expressed in the state-space syntax $\mathtt{V}_s$, which we claim is only for convenience: it is possible to express arbitrary $\mathtt{V}_s$ programs in Verilog syntax. We now make precise the correspondence between Verilog and $\mathtt{V}_s$.

5.1 From State-Space to Verilog

The critical property of a $\mathtt{V}_s$ program is the explicit representation of both the store and the simulation cycle. By tagging each thread with its execution status, it is easy to enforce the order implied by the simulation model. The translation to Verilog uses delays and guards to enforce this ordering:

$$\begin{array}{lll} V\ \ : P_s \longrightarrow P_v & V_d : d_v \times u \longrightarrow d_v & V_{m_1} : \mathtt{m}_1\{\phi\} \longrightarrow t_v \\ V_{m_2}\ : \mathtt{m}_2\{\phi^+\} \times u \longrightarrow t_v & V_t : t_s \times u \longrightarrow t_v & V_n\ : \mathtt{nbu}\{\theta\} \longrightarrow t_v \end{array}$$

$$\begin{array}{ll} V(d_v\ \ \mathtt{m}_1\{\phi\}\ \mathtt{m}_2\{\phi^+\}\ \ {t_s}^n\ \ \mathtt{nbu}\{\theta\}) & \\ & = \mathtt{module\ m;}\ \ (V_d(d_v)\ \omega)\ \ V_{m_1}(\mathtt{m}_1\{\phi\})\ (V_{m_2}(\mathtt{m}_1\{\phi^+\})\ \omega) \\ & \quad (V_t(t_s)\ \omega)^n\ \ V_n(\mathtt{nbu}\{\theta\}) \\ & \quad \mathtt{endmodule} \\ & \qquad \bullet \text{ where } \omega \notin d_v \\ V_d(d_v)\ \omega & = d_v\ \mathtt{event}\ \omega\mathtt{;} \\ V_{m_1}(\mathtt{m}_1\{\phi\}) & = \mathtt{initial}\ \phi \\ V_{m_2}(\mathtt{m}_2\{\phi^+\})\ \omega & = \mathtt{initial\ \#0\#0\ begin}\ \phi^+\ \ \mathtt{->}\omega\mathtt{;\ end} \\ V_t(\mathtt{at}\{s\})\ \omega & = \mathtt{initial\ @(}\omega\mathtt{)}\ s \\ V_t(\mathtt{zd}\{s\})\ \omega & = \mathtt{initial\ @(}\omega\mathtt{)\#0}\ s \\ V_t(\mathtt{fd}\{s\})\ \omega & = \mathtt{initial}\ s \\ V_t(\mathtt{fg}\{s\})\ \omega & = \mathtt{initial\ \#0}\ s \\ V_n(\mathtt{nbu}\{\theta\}) & = \mathtt{initial}\ \theta \end{array}$$

This translation ensures an ordering among the threads corresponding to that required by the standard [13]. The idea here is to ensure that the contents of memory are updated appropriately and that every non-active thread is first scheduled to the appropriate phase of the cycle; all of this must be done before the active threads can execute.

To accomplish this, the initialization of the first copy of the store, the future-delayed statements, and the non-blocking updates must be executed first. Note that the execution of the future-delayed statements just causes the statements to be scheduled later and the execution of the non-blocking updates just causes the assignments to be scheduled later. Next, the threads with a 0-delay are allowed to run: these include all the future-guarded threads. Again the execution of these threads blocks immediately. Finally the initialization of the second copy of the store can be executed, which triggers the added event ω, enabling the remaining threads.

The addition of the ω event trigger to $\mathtt{m}_2$ and of $\mathtt{@}(\omega)$ to the bodies of *active* and *delayed* threads is a change from previous work [9]. It arises from the requirements of capturing edges in the state, as represented in ϕ^+. Any edges that exist in ϕ^+ correspond to event triggers and assignments that occurred *within the current* δ. In other words, these are the results of statements from threads which were executing concurrently with the current `at`'s, and were (nondeterministically) selected first.

5.2 Correctness

Our translation from state-space syntax to Verilog uses delays and guards judiciously to enforce ordering constraints on the execution of threads. The complexity of the translation, however, raises the question of its correctness.

Intuitively the problem could be that our translations to and from state-space syntax are not consistent with the axioms. Another way to look at the problem is that the axioms are expressed on equivalence classes of terms that are identified by the translations to and from state-space syntax. But if two different representatives of the equivalence class were equal to two terms in different equivalence classes, our theory would be inconsistent.

Formally we need to prove that the translations to and from state-space syntax yield provably equal terms. In fact this is a special case of an *equational correspondence* which is the standard way of relating two syntactic theories [3,7,17]. To see this, let λ_v be the syntactic theory for Verilog and let λ_s be the syntactic theory for $\mathtt{V}_s$. Let P_v and $P_v{}'$ be programs in Verilog, P_s and $P_s{}'$ programs in $\mathtt{V}_s$, and define S and V as above.

Theorem 5.1 *There is an equational correspondence between λ_v and λ_s. In other words:*

$$\begin{array}{rl}
i. & \lambda_s \vdash S \circ V(P_s) \stackrel{ax}{=} P_s \\
ii. & \lambda_v \vdash V \circ S(P_v) \stackrel{ax}{=} P_v \\
iii. & \lambda_s \vdash P_s \stackrel{ax}{=} P_s{}' \Longrightarrow \lambda_v \vdash V(P_s) \stackrel{ax}{=} V(P_s{}') \\
iv. & \lambda_v \vdash P_v \stackrel{ax}{=} P_v{}' \Longrightarrow \lambda_s \vdash S(P_v) \stackrel{ax}{=} S(P_v{}')
\end{array}$$

Before discussing the proofs, observe that λ_s is just the set of axioms given in this paper for $\mathtt{V}_s$. But $\mathtt{V}_s$ itself is considered a mere notational device for Verilog. Thus, the syntactic theory for Verilog, λ_v, is defined in terms of λ_s:

$$\lambda_v \vdash P_v \overset{ax}{=} P_v' \qquad \text{if} \qquad \lambda_s \vdash S(P_v) \overset{ax}{=} S(P_v')$$

The result is that our proof burden is greatly simplified. Statement (iv) becomes trivially true, for example. Further, the task of proving statements (i)–(iii) collapses into a proof of (i) only.

Proof (outline) of (i)

We let $P_s = d_v\ \mathtt{m}_1\{\phi\}\ \mathtt{m}_2\{\phi^+\}\ \ t_s{}^n\ \ \mathtt{nbu}\{\theta\}$ be a program in $\mathtt{V}_s$. Without loss of generality, we can assume that $t_s{}^n = \mathtt{at}\{s\}^p\ \mathtt{zd}\{s\}^q\ \mathtt{fd}\{\#n\ s\}^r\ \mathtt{fg}\{@(g)\ s\}^s$, where $n = p + q + r + s$ (otherwise, we can apply (ax_ord) to re-order the source code). The proof then proceeds by giving the construction of derivations by which both $S \circ V(P_s)$ and P_s are rewritten to a common term, $P_s{}'$. Intuitively, this derivation corresponds to an execution of $S \circ V(P_s)$ that "restores" the state of each thread.

□

Once this is in place, (ii) is established as an immediate consequence of (i). In fact, (i) also implies (iii):

Proof of (iii)

By definition, this is the same as

$$\lambda_s \vdash P_s \overset{ax}{=} P_s{}' \Longrightarrow \lambda_s \vdash S(V(P_s)) \overset{ax}{=} S(V(P_s{}'))$$

We suppose that $\lambda_s \vdash P_s \overset{ax}{=} P_s{}'$, according to some derivation D_1. By (i), we also have derivations D_2 and D_3, which prove $P_s \overset{ax}{=} S(V(P_s))$ and $P_s{}' \overset{ax}{=} S(V(P_s{}'))$ (respectively). The conclusion then follows from D_1, D_2, D_3, and the symmetry and transitivity of $\overset{ax}{=}$.

□

6 Automatic Generation of a Verilog Simulator

The syntactic approach to operational specification suffers a disavantage from the complexity of the resulting derivations needed to express a program's behavior. In our theory, the evaluation of even trivial programs requires very lengthy rewrites (the proof of 5.1 (i) itself runs to five pages). From a pragmatic standpoint, manual evaluation of a real program is therefore impossible; some form of automatic support is essential.

To this end, we have input our theory to a tool that generates a syntactic term-rewriting machine from a given specification [21]. We found that the implementation of the theory required the addition of a few simple CAML functions to implement the context-sensitive side-conditions. Further, the resulting term-rewriting machine had a tendency to apply (ax_ord) in an apparently endless loop, and we therefore found it necessary to direct the machine's proof strategy away from this axiom.

Otherwise, the implementation was almost perfectly straightforward, resulting in a correct-by-construction simulator that was completed and debugged in less than three days' time.

7 Conclusion

With some extensions to existing operational specification techniques, we have constructed a syntactic theory for a substantial portion of the behavioral subset of Verilog, including an explicit description of its standard discrete event model. In the process, our work clarifies the nature of an established informal practice in the construction of syntactic theories, namely the use of an abstract state-space syntax. The consistency of a state-space syntax with the syntactic theory entails a proof obligation which, for industrial languages, is likely to be non-trivial. We have shown, however, that this obligation is just a special case of an equational correspondence proof.

There remain certain Verilog constructs which have thus far defied attempts at encoding. In particular, we have not yet found any axiomatic description of Verilog's low-level (structural) subset. This includes the `wire` data structure, and certain hardware-like behaviors such as continuous assignment and inertial delay. The main obstacle seems to be the lack of an effective way to represent state in a program which includes `wire` variables. Consider, for example, the `struc_sr_test` example from section 3.2 and the pseudo-code resulting from the transformation $V \circ S$:

```
module struc_sr_test;
  wire q; reg r,s; event ω;
  initial  begin q = x; s = x; r = x; end
  initial #0#0  begin q = x; s = x; r = x; ->ω; end
  assign q =  ~(r |~(s | q));
    ⋮
  initial  ;   (no pending nb updates)
endmodule
```

This illustrates two problems. The first is that the traditional representation of state by a sequence of assignments cannot be used with variables of `wire` type. Procedural assignments are only allowed to `reg` variables: the statement q=`x` is an error. On the other hand, one cannot simply eliminate q by substituting with the RHS of the continuous assignment, as this would result in an infinite expansion. A satisfactory theory for constructs such as this remains an open problem.

Even more interesting is the challenge of relating such a theory with the existing description for behavioral Verilog. A solution to this would be of serious interest within industry, as it would provide a method of correct-by-construction synthesis of circuitry from high-level specifications.

In some cases, we have purposefully excluded certain behavioral constructs. For example, the full language specification allows event controls to guard on arbitrary expressions. By contrast, we have chosen to restrict the allowable controls to single variables only. With the introduction of arbitrary guard expressions, the LRM requirements of full statement/expression interleaving and guaranteed event control release are mutually exclusive: it is not possible to guarantee event control release on an expression that can change value in the middle of its evaluation.

Finally, the most immediate task that remains in our work is the proof of several meta-properties about our theory. We have not yet shown the consistency of the axioms nor any normalization properties. Given their complexity, we suspect that some form of automatic support will be required for this.

References

[1] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. *J. Functional Programming 7*, 3 (1997).

[2] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Pr., New York, pp. 233–246.

[3] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, 1984.

[4] Boehm, H. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 637–655.

[5] Cheng, S., and Brayton, R. Synthesizing multi-phase HDL programs. In *1996 IEEE International Verilog HDL Conference* (Washington, 1996), IEEE Press, pp. 67–76.

[6] Cheng, S., Brayton, R., York, G., Yelick, K., and Saldanha, A. Compiling Verilog into timed finite state machines. In *1995 IEEE International Verilog HDL Conference* (Washington, 1995), IEEE Press, pp. 32–39.

[7] Curry, H. B., and Feys, R. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1958.

[8] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci. 102* (1992), 235–271. Tech. Rep. 89-100, Rice University.

[9] Fiskio-Lasseter, J. A formal description of behavioral Verilog based on axiomatic semantics. M.Sc Thesis, Department of Computer and Information Science, University of Oregon, 1998. Tech. Rep. CIS-TR-98-04.

[10] Gordon, M. The semantic challenge of Verilog HDL. In the *IEEE Symposium on Logic in Computer Science* (June 1995), IEEE Computer Society Press, Los Alamitos, Calif. Revised version available (as of 08-24-99) on the World-Wide Web at `http://www.cl.cam.uk/users/mjcg/Verilog/V.ps.Z`.

[11] Gordon, M., and Ghosh, A. Language independent RTL Semantics. Unpublished manuscript, available (as of 08-24-99) on the World-Wide Web at `http://www.cl.cam.uk/users/mjcg/Verilog/V/V.html`, 1998.

[12] Hua, X., and Zhang, H. Axiomatic semantics of a hardware specification language. In *Second Great Lakes Symposium on VLSI* (Los Alamitos, CA, 1991), IEEE Press, pp. 183–190.

[13] IEEE, Ed. *Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Computer Society Press, Los Alamitos, Calif., 1996. IEEE Standard 1364-1995.

[14] Mason, I., and Talcott, C. L. Equivalence in functional languages with effects. *J. Functional Programming 1*, 3 (July 1991), 287–327.

[15] Pace, G., and He, J. Formal Reasoning with Verilog HDL. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems* (Marstrand, Sweden, June 1998).

[16] Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci. 1* (1975), 125–159.

[17] Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp Symbol. Comput. 6*, 3/4 (1993), 289–360. Also in the *ACM Conference on Lisp and Functional Programming* (1992) and Tech. Rep. 92-180, Rice University.

[18] Sasaki, H. A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine. In *Design, Automation, and Test in Europe (DATE99)* (1999), pp. 353–357.

[19] Talcott, C. L. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-Type Computation.* PhD thesis, Stanford University, 1985.

[20] Van Tassel, J. P. A formalisation of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications* (North-Holland, 1993), L. Claesen and M. Gordon, Eds., Elsevier Science Publishers B.V., pp. 359–374.

[21] Xiao, Y., Mauny, M., Remy, D., and Leroy, X. A prototype of compiling semantics. Unpublished manuscript, 1999.